About This Book

The C language is a general-purpose, function-oriented programming language that you can use to create applications quickly and easily. C provides high-level control statements and data types similar to other structured programming languages as well as many of the benefits of a low-level language. Portable code is easily written in the C language.

This book describes the library functions provided by *The IBM Developer's Toolkit for OS/2 Warp Version 4* (referred to in this book as *The Developer's Toolkit*) product. Many of the functions are defined by the following language standards:

- The American National Standards Institute C Standard and International Standards Organization, ANSI/ISO 9899-1990[1992], and the amendment ISO/IEC 9899:1990/Amendment 1:1993(E)
- The ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard
- The X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.

Unless explicitly indicated otherwise, all of the library functions described in this book are also available to C++ programs.

Who Should Read This Book

This book is written for application programmers who want to use the functions and macros provided by *The Developer's Toolkit* to develop and run C and C++ applications on the Operating System/2 (OS/2) platform. It assumes you have a working knowledge of the C programming language and the OS/2 Warp Version 4 operating system.

Portability Considerations

If you will be using *The Developer's Toolkit* to develop applications that will also be compiled and run on other systems, you should refer to the current language standards described in Preface above.

The language level is given for each function in Library Functions. When creating programs to conform strictly to language standards such as ANSI/ISO or POSIX, do **not** use the functions described as extensions in this book.

The C Library

This chapter summarizes the available C library functions and indicates where in this book each is described. It also briefly describes what the function does. Each library function is listed according to the type of function it performs.

Summary of Library Functions

The functions described in this book are listed according to the following tasks they perform:

Error Handling

Function	Header File	Link to Function	Descriptio
assert	"assert.h"	assert	Prints

			diagnostic messages.
atexit	"stdlib.h"	atexit	Registers a function to be executed at program termination.
clearerr	"stdio.h"	clearerr	Resets error indicators.
ferror	"stdio.h"	ferror	Tests the error indicator for a specified stream.
_matherr	"math.h"	_matherr	Processes errors generated by the functions in the math library.
perror	"stdio.h"	perror	Prints an error message to stderr.
raise	"signal.h"	raise	Initiates a signal.
_set_crt_msg_handle	"stdio.h"	_set_crt_msg_handle	Changes the file handle to which run-time messages are sent.
signal	"signal.h"	signal	Allows handling of an interrupt signal from the operating system.
strerror	"string.h"	strerror	Sets pointer to system error message.

_strerror	"string.h"	_strerror	Tests for system error.

Process Control

Function	Header File	Link to Function	Descriptio
_beginthread	"stdlib.h"" process.h"	_beginthread	Creates a new thread.
_cwait	"process.h"	_cwait	Delays the completion of a parent process until a child process ends.
_endthread	"stdlib.h"" process.h"	_endthread	Terminates a thread.
execl - _execvpe	"process.h"	execl - _execvpe	Load and run child processes.
_exit	"stdlib.h"" process.h"	_exit	Ends the calling process without calling other functions.
getpid	"process.h"	getpid	Gets the process identifier that identifies the calling process.
_onexit	"stdlib.h"	_onexit	Records the address of a function to call when the

			program ends.
putenv	"stdlib.h"	putenv	Adds new environment variables or modifies the values of those already existing.
_searchenv	"stdlib.h"	_searchenv	Searches a specified environment for a target file.
_spawnl - _spawnvpe	"process.h"	_spawnl - _spawnvpe	Start and run child processes.
_threadstore	"stdlib.h"	_threadstore	Accesses a thread-specific storage space.

File and Directory Management

Function	Header File	Link to Function	Descriptio
chdir	"direct.h"	chdir	Changes the current working directory to a specified directory.
_chdrive	"direct.h"	_chdrive	Changes the current working drive to a specified drive.

fstat	"sys\stat.h"	fstat	Gets and stores information about the open file.
_fullpath	"stdlib.h"	_fullpath	Gets and stores the full path name of a given partial path.
_getcwd	"direct.h"	_getcwd	Gets the full path name of the current working directory.
_getdcwd	"direct.h"	_getdcwd	Gets the full path name for the current directory of a specified drive.
_getdrive	"direct.h"	_getdrive	Returns an integer corresponding to the letter representing the current drive.
_makepath	"stdlib.h"	_makepath	Creates a single path name.
mkdir	"direct.h"	mkdir	Creates a new directory.
rmdir	"direct.h"	rmdir	Deletes a directory.
_splitpath	"stdlib.h"	_splitpath	Decomposes a

			path name into its four components.
stat	"sys\stat.h"	stat	Stores information about a file or directory in a structure.

Searching and Sorting

Function	Header File	Link to Function	Descriptio
bsearch	"stdlib.h"" search.h"	bsearch	Performs a binary search of a sorted array.
lfind	"search.h"	lfind - lsearch	Performs a linear search for a value in an array.
lsearch	"search.h"	lfind - lsearch	Performs a linear search for a value in an array.
qsort	"stdlib.h"" search.h"	qsort	Performs a quick sort on an array of elements.

Regular Expressions

Function	Header File	Link to Function	Descriptio

regcomp	"regex.h"	regcomp	Compiles a source regular expression into an executable version.
regerror	"regex.h"	regerror	Finds the descriptio for the error code for a regular expression
regexec	"regex.h"	regexec	Compares a string with a regular expression
regfree	"regex.h"	regfree	Frees the memory for a regular expression removing the compiled regular expression

---

Mathematical

Function	Header File	Link to Function	Descriptio
abs	"stdlib.h"	abs	Calculates the absolute value of an integer.
ceil	"math.h"	ceil	Calculates the double value representing the smallest integer that is greater

			than or equal to a number.
div	"stdlib.h"	div	Calculates the quotient and remainder of an integer.
erf	"math.h"	erf - erfc	Calculates the error function.
erfc	"math.h"	erf - erfc	Calculates the error function for large numbers.
exp	"math.h"	exp	Calculates an exponential function.
fabs	"math.h"	fabs	Calculates the absolute value of a floating-point number.
floor	"math.h"	floor	Calculates the double value representing the largest integer that is less than or equal to a number.
fmod	"math.h"	fmod	Calculates the floating point remainder of one argument divided by another.

frexp	"math.h"	frexp	Separates a floating-point number into its mantissa and exponent.
gamma	"math.h"	gamma	Calculates the gamma function.
hypot	"math.h"	hypot	Calculates the hypotenuse
labs	"stdlib.h"	labs	Calculates the absolute value of a long integer.
ldexp	"math.h"	ldexp	Multiplies a floating-point number by an integral power of 2.
ldiv	"stdlib.h"	ldiv	Calculates the quotient and remainder of a long integer.
log	"math.h"	log	Calculates natural logarithm.
log10	"math.h"	log10	Calculates base 10 logarithm.
max	"stdlib.h"	max	Compares two values and returns the larger of the two.
min	"stdlib.h"	min	Compares

			two values and returns the smaller of the two.
modf	"math.h"	modf	Calculates the signed fractional portion of the argument.
pow	"math.h"	pow	Calculates the value of an argument raised to a power.
sqrt	"math.h"	sqrt	Calculates the square root of a number.

Trigonometric Functions

Function	Header File	Link to Function	Descriptio
acos	"math.h"	acos	Calculates the arccosine.
asin	"math.h"	asin	Calculates the arcsine.
atan	"math.h"	atan - atan2	Calculates the arctangent
atan2	"math.h"	atan - atan2	Calculates the arctangent
cos	"math.h"	cos	Calculates the cosine.

cosh	"math.h"	cosh	Calculates the hyperbolic cosine.
sin	"math.h"	sin	Calculates the sine.
sinh	"math.h"	sinh	Calculates the hyperbolic sine.
tan	"math.h"	tan	Calculates the tangent.
tanh	"math.h"	tanh	Calculates the hyperbolic tangent.

Bessel Functions

Function	Header File	Link to Function	Descriptio
"_j0"	"math.h"	bessel	0 order differential equation of the first kind.
"_j1"	"math.h"	bessel	1st order differential equation of the first kind.
"_jn"	"math.h"	bessel	nth order differential equation of the first kind.
"_y0"	"math.h"	bessel	0 order differential equation of the second kind.

"_y1"	"math.h"	bessel	1st order differential equation of the second kind.
"_yn"	"math.h"	bessel	nth order differential equation of the second kind.

Date, Time, and Monetary Manipulation

Function	Header File	Link to Function	Descriptio
asctime	"time.h"	asctime	Converts time stored as a structure to a character string in storage.
clock	"time.h"	clock	Determines processor time.
ctime	"time.h"	ctime	Converts time stored as a long value to a character string.
difftime	"time.h"	difftime	Calculates the difference between two times.
_ftime	"sys\timeb.h"	_ftime	Obtains the current time and stores it

			in a structure.
gmtime	"time.h"	gmtime	Converts time to Coordinated Universal Time structure.
localtime	"time.h"	localtime	Converts time to local time.
mktime	"time.h"	mktime	Converts local time into calendar time.
_strdate	"time.h"	_strdate	Stores the current date in a buffer.
strfmon	"monetary.h"	strfmon	Converts a monetary value to a formatted string.
strftime	"time.h"	strftime	Converts time to a multibyte character string.
strptime	"time.h"	strptime	Converts time stored as a character string to a structure.
_strtime	"time.h"	_strtime	Copies the current time into a buffer.
time	"time.h"	time	Returns the time in seconds.
tzset	"time.h"	tzset	Changes

			the time zone and daylight saving time zone values.
utime	"sys\utime.h"	utime	Sets the modification time for a file.
wcsftime	"wchar.h"	wcsftime	Converts the time and date to a wide character string.

Messages

Function	Header File	Link to Function	Descriptio
catclose	"nl_types.h"	catclose	Closes a specified message catalog.
catgets	"nl_types.h"	catgets	Retrieves a message from a catalog.
catopen	"nl_types.h"	catopen	Opens a specified message catalog.

Type Conversion

Function	Header File	Link to Function	Descriptio
atof	"stdlib.h"	atof	Converts a character string to a floating-

			point value.
atoi	"stdlib.h"	atoi	Converts a character string to an integer.
atol	"stdlib.h"	atol	Converts a character string to a long integer.
_atold	"stdlib.h"" math.h"	_atold	Converts a character string to a long double value.
_ecvt	"stdlib.h"	_ecvt	Converts a floating-point number to a character string.
_fcvt	"stdlib.h"	_fcvt	Converts a floating-point number to a character string, rounding according to the FORTRAN F format.
_gcvt	"stdlib.h"	_gcvt	Converts a floating-point value to a character string, rounding according to the FORTRAN F or FORTRAN E formats.
_itoa	"stdlib.h"	_itoa	Converts the

			digits of an integer to a character string.
_ltoa	"stdlib.h"	_ltoa	Converts the digits of a long integer to a character string.
strtod	"stdlib.h"	strtod	Converts a character string to a double value.
strtol	"stdlib.h"	strtol	Converts a character string to a long integer.
strtold	"stdlib.h"	strtold	Converts a character string to a long double value.
strtoul	"stdlib.h"	strtoul	Converts a string to an unsigned long integer.
_ultoa	"stdlib.h"	_ultoa	Converts the values of a long value to a character string.
wcstod	"wchar.h"	wcstod	Converts a wide character string to a double value.
wcstol	"wchar.h"	wcstol	Converts a wide character string to

			a long integer.
wcstoul	"wchar.h"	wcstoul	Converts a wide character string to an unsigned long integer.
wctob	"wchar.h"	wctob	Converts a wide character to a single-byte character.

Multibyte and Wide-Character Type Conversion

Function	Header File	Link to Function	Descriptio
mbstowcs	"stdlib.h"	mbstowcs	Converts a multibyte character string to a wide character (wchar_t) string.
mbtowc	"stdlib.h"	mbtowc	Converts a multibyte character to a wide character (wchar_t).
wcstombs	"stdlib.h"	wcstombs	Converts a wide character (wchar_t) string to a multibyte character string.
wctomb	"stdlib.h"	wctomb	Converts a wide character (wchar_t) to a

multibyte character.

Stream Input/Output

This section describes the following input/output functions:

- Formatted Input/Output
- Character and String Input/Output
- Wide Character and String Input/Output
- Direct Input/Output
- File Positioning
- File Access
- File Operations

Formatted Input/Output

Function	Header File	Link to Function	Descriptio
fprintf	"stdio.h"	fprintf	Formats and prints characters to the output stream.
fscanf	"stdio.h"	fscanf	Reads data from a stream into locations given by arguments.
printf	"stdio.h"	printf	Formats and prints characters to stdout.
scanf	"stdio.h"	scanf	Reads data from stdin into locations given by arguments.

sprintf	"stdio.h"	sprintf	Formats and writes characters to a buffer.
sscanf	"stdio.h"	sscanf	Reads data from a buffer into locations given by arguments.
vfprintf	"stdarg.h"" stdio.h"	vfprintf	Formats and prints characters to the output stream using a variable number of arguments.
vprintf	"stdarg.h"" stdio.h"	vprintf	Formats and writes characters to stdout using a variable number of arguments.
vsprintf	"stdarg.h"" stdio.h"	vsprintf	Formats and writes characters to a buffer using a variable number of arguments.

Character and String Input/Output

Function	Header File	Link to Function	Descriptio
fgetc	"stdio.h"	fgetc	Reads a character from a specified input stream.
fgets	"stdio.h"	fgets	Reads a string from a specified input stream.
fputc	"stdio.h"	fputc	Prints a character to a specified output stream.
fputs	"stdio.h"	fputs	Prints a string to a specified output stream.
getc	"stdio.h"	getc - getchar	Reads a character from a specified input stream.
getchar	"stdio.h"	getc - getchar	Reads a character from stdin.
gets	"stdio.h"	gets	Reads a line from stdin.
putc	"stdio.h"	putc - putchar	Prints a character to a specified output stream.
putchar	"stdio.h"	putc - putchar	Prints a character to stdout.
puts	"stdio.h"	puts	Prints a string to stdout.
ungetc	"stdio.h"	ungetc	Pushes a character back onto

			a specified input stream.

Wide Character and String Input/Output

Function	Header File	Link to Function	Descriptio
fgetwc	"stdio.h"" wchar.h"	fgetwc	Reads a wide character from a specified input stream.
fgetws	"stdio.h"" wchar.h"	fgetwc	Reads a wide string from a specified input stream.
fputwc	"stdio.h"" wchar.h"	fputwc	Writes a wide character to a specified output stream.
fputws	"stdio.h"" wchar.h"	fputws	Writes a wide character string to a specified output stream.
getwc	"stdio.h"" wchar.h"	getwc	Reads a wide character from a specified input stream.
getwchar	"wchar.h"	getwchar	Reads a wide character from stdin.
putwc	"stdio.h""	putwc	Writes a

	wchar.h"		wide character to a specified output stream.
putwchar	"wchar.h"	putwchar	Writes a wide character to stdout.
ungetwc	"stdio.h"" wchar.h"	ungetwc	Pushes a wide character back onto a specified input stream.

Direct Input/Output

Function	Header File	Link to Function	Descriptio
clearerr	"stdio.h"	clearerr	Resets error indicators.
feof	"stdio.h"	feof	Tests end-of-file indicator for stream input.
ferror	"stdio.h"	ferror	Tests the error indicator for a specified stream.
fread	"stdio.h"	fread	Reads items from a specified input stream.
fwrite	"stdio.h"	fwrite	Writes items to a specified output

stream.

File Positioning

Function	Header File	Link to Function	Descriptio
fgetpos	"stdio.h"	fgetpos	Gets the current position of the file pointer.
fseek	"stdio.h"	fseek	Moves the file pointer to a new location.
fsetpos	"stdio.h"	fsetpos	Moves the file pointer to a new location.
ftell	"stdio.h"	ftell	Gets the current position of the file pointer.
lseek	"io.h"	lseek	Moves a file pointer to a new location.
rewind	"stdio.h"	rewind	Repositions the file pointer to the beginning of the file.

File Access

Function	Header File	Link to Function	Descriptio
fclose	"stdio.h"	fclose	Closes a specified stream.
_fcloseall	"stdio.h"	_fcloseall	Closes all open streams, except the standard streams.
fdopen	"stdio.h"	fdopen	Associates an input or output stream with a file.
fflush	"stdio.h"	fflush	Causes the system to write the contents of a buffer to a file.
_flushall	"stdio.h"	_flushall	Writes the contents of buffers to files.
fopen	"stdio.h"	fopen	Opens a specified stream.
freopen	"stdio.h"	freopen	Closes a file and reassigns a stream.
setbuf	"stdio.h"	setbuf	Allows control of buffering.
_setmode	"io.h"	_setmode	Sets the translation mode of a file.
setvbuf	"stdio.h"	setvbuf	Controls buffering and buffer size for a specified stream.

File Operations

Function	Header File	Link to Function	Descriptio
fileno	"stdio.h"	fileno	Determines the file handle.
remove	"stdio.h"	remove	Deletes a specified file.
rename	"stdio.h"	rename	Renames a specified file.
_rmtmp	"stdio.h"	_rmtmp	Closes and deletes temporary files.
tempnam	"stdio.h"	tempnam	Creates a temporary file name in another directory.
tmpfile	"stdio.h"	tmpfile	Creates a temporary file and returns a pointer to that file.
tmpnam	"stdio.h"	tmpnam	Produces a temporary file name.
unlink	"stdio.h"	unlink	Deletes a file.

Low-Level Input/Output

This section describes the following input/output functions:

Port Input/Output

Function	Header File	Link to Function	Descriptio
umask	"io.h"	umask	Sets the file permission mask of the executing process environment.

Character and String Input/Output

Function	Header File	Link to Function	Descriptio
_cgets	"conio.h"	_cgets	Reads a string from the keyboard into locations given by arguments.
_cprintf	"conio.h"	_cprintf	Formats and sends a series of characters and values to the screen.
_cputs	"conio.h"	_cputs	Writes a string directly to the screen.

_cscanf	"conio.h"	_cscanf	Reads data from the keyboard into locations given by arguments.
_getch	"conio.h"	_getch - _getche	Reads a single character from the keyboard.
_getche	"conio.h"	_getch - _getche	Reads a single character from the keyboard and displays it.
_kbhit	"conio.h"	_kbhit	Tests if a key has been pressed on the keyboard.
_putch	"conio.h"	_putch	Writes a character to the screen.
_ungetch	"conio.h"	_ungetch	Pushes a character back to the keyboard.

Direct Input/Output

Function	Header File	Link to Function	Descriptio
read	"io.h"	read	Reads bytes from a file into a buffer.
write	"io.h"	write	Writes bytes from a buffer into a

file.

File Positioning

Function	Header File	Link to Function	Descriptio
__eof	"io.h"	__eof	Determines whether the file pointer has reached the end of the file.
_tell	"io.h"	_tell	Gets the current position of a file pointer.

File Access

Function	Header File	Link to Function	Descriptio
access	"io.h"	access	Determines whether the given file exists and whether you can gain access to it.
chmod	"io.h"	chmod	Changes the permission setting of a file.
close	"io.h"	close	Closes a file

			associated with the handle.
creat	"io.h"	creat	Creates a new file or opens and truncates an existing file.
dup	"io.h"	dup	Associates a second file handle with an open file.
dup2	"io.h"	dup2	Associates a second file handle, with possibly different attributes, with an open file.
isatty	"io.h"	isatty	Determines whether the handle is associated with a character device.
open	"io.h"	open	Opens a file and prepares it for subsequent reading and writing.
_sopen	"io.h"	_sopen	Opens a file and prepares it for subsequent shared reading

			or writing.

File Operations

Function	Header File	Link to Function	Descriptio
_chsize	"io.h"	chsize	Lengthens or cuts off the file to a specified length.
_filelength	"io.h"	_filelength	Returns the length of a file.

Handling Argument Lists

Function	Header File	Link to Function	Descriptio
va_arg	"stdarg.h"	va_arg - va_end - va_start	Allows access to variable number of function arguments.
va_end	"stdarg.h"	va_arg - va_end - va_start	Allows access to variable number of function arguments.
va_start	"stdarg.h"	va_arg - va_end - va_start	Allows access to variable number of function arguments.
vfprintf	"stdarg.h"" stdio.h"	vfprintf	Formats and

			prints characters to the output stream using a variable number of arguments.
vprintf	"stdarg.h"" stdio.h"	vprintf	Formats and writes characters to stdout using a variable number of arguments.
vsprintf	"stdarg.h"" stdio.h"	vsprintf	Formats and writes characters to a buffer using a variable number of arguments.

Pseudorandom Numbers

Function	Header File	Link to Function	Descriptio
rand	"stdlib.h"	rand	Returns a pseudorand m integer.
srand	"stdlib.h"	srand	Sets the starting point for pseudorand m numbers.

Dynamic Memory Management

This section describes the following input/output functions:

- Allocating and Freeing Memory
- Heap Information and Checking
- Heap Creation and Management

Allocating and Freeing Memory

Function	Header File	Link to Function	Descriptio
calloc	"stdlib.h"" malloc.h"	calloc	Reserves storage space for an array and initializes the values of all elements to 0.
free	"stdlib.h"" malloc.h"	free	Releases memory blocks.
_heapmin	"stdlib.h"" malloc.h"	_heapmin	Returns all unused memory from the run-time heap to the operating system.
malloc	"stdlib.h"" malloc.h"	malloc	Allocates memory blocks.
realloc	"stdlib.h"" malloc.h"	realloc	Changes storage size allocated for an object.

Heap Information and Checking

Function	Header File	Link to Function	Descriptio
_heapset	"malloc.h"	_heapset	Validates all allocated and freed objects on the default heap, and sets all free memory to a specified value.
_heapchk	"malloc.h"	_heapchk	Checks all allocated and freed objects on the default storage heap for minimal consistency.
_heap_walk	"malloc.h"	_heap_walk	Returns information about allocated and freed objects on the default heap.
_mheap	"umalloc.h"	_mheap	Finds out which heap an object was allocated from.
_msize	"stdlib.h"" malloc.h"	_msize	Returns the size of an allocated block.
_uheapchk	"umalloc.h"	_uheapchk	Validates all allocated and freed objects on a specified heap.
_uheapset	"umalloc.h"	_uheapset	Validates all allo-

			cated and freed objects on a specified heap, and sets all free memory to a specified value.
_uheap_walk	"umalloc.h"	_uheap_walk	Returns information about allocated and freed objects on a specified heap.
_ustats	"umalloc.h"	_ustats	Gets information about a specified heap.

Heap Creation and Management

Function	Header File	Link to Function	Descriptio
_uaddmem	"umalloc.h"	_uaddmem	Adds memory to a specified heap.
_uclose	"umalloc.h"	_uclose	Closes a heap so it can no longer be used.
_ucreate	"umalloc.h"	_ucreate	Creates a heap of memory.
_udefault	"umalloc.h"	_udefault	Changes the memory heap used as the default

			heap.
_udestroy	"umalloc.h"	_udestroy	Destroys a heap.
_uopen	"umalloc.h"	_uopen	Opens a heap so it can be used.

Memory Objects

Function	Header File	Link to Function	Descriptio
memchr	"string.h"" memory.h"	memchr	Searches a buffer for the first occurrence of a given character.
memcmp	"string.h"" memory.h"	memcmp	Compares two buffers.
memcpy	"string.h"" memory.h"	memcpy	Copies a buffer.
memicmp	"string.h"" memory.h"	memicmp	Compares two buffers without regard to case.
memmove	"string.h"" memory.h"	memmove	Moves a buffer.
memset	"string.h"" memory.h"	memset	Sets a buffer to a given value.
swab	"stdlib.h"	swab	Copies bytes from a specified source and swaps each pair of adjacent bytes.

Environment Interaction

Function	Header File	Link to Function	Descriptio
abort	"stdlib.h"" process.h"	abort	Terminates a program abnormally.
exit	"stdlib.h"" process.h"	exit	Ends a program normally.
getenv	"stdlib.h"	getenv	Searches environment variables for a specified variable.
longjmp	"setjmp.h"	longjmp	Restores a stack environment.
setjmp	"setjmp.h"	setjmp	Saves a stack environment.
system	"stdlib.h"" process.h"	system	Passes a string to the operating system's command interpreter.

Setting and Querying Locale

Function	Header File	Link to Function	Descriptio
iconv	"iconv.h"	iconv	Converts characters from one

			codeset to another.
iconv_close	"iconv.h"	iconv_close	Deletes the conversion descriptor created by iconv_open
iconv_open	"iconv.h"	iconv_open	Creates a conversion descriptor for iconv to use in converting characters.
localeconv	"locale.h"	localeconv	Queries the numeric formatting conventions for the current locale.
nl_langinfo	"langinfo.h"" nl_types.h"	nl_langinfo	Retrieves requested information for the current locale.
setlocale	"locale.h"	setlocale	Changes or queries the locale.
wctype	"wctype.h"	wctype	Returns the handle for a character class or property.

String and Character Collating

Function	Header File	Link to Function	Descriptio
strcoll	"string.h"	strcoll	Compares two strings based on the collating elements in the current locale.
wcscoll	"wchar.h"	wcscoll	Compares two wide character strings based on the collating elements for the current locale.

String Operations

Function	Header File	Link to Function	Descriptio
strcat	"string.h"	strcat	Concatenates two strings.
strchr	"string.h"	strchr	Locates the first occurrence of a specified character in a string.
strcmp	"string.h"	strcmp	Compares the value of two strings.
strcmpi	"string.h"	strcmpi	Compares two strings without sensitivity to case.
strcoll	"string.h"	strcoll	Compares

			two strings based on the collating elements for the current locale.
strcpy	"string.h"	strcpy	Copies one string into another.
strcspn	"string.h"	strcspn	Finds the length of the first substring in a string of characters not in a second string.
strdup	"string.h"	strdup	Reserves storage space for the copy of a string.
stricmp	"string.h"	stricmp	Compares two strings without sensitivity to case.
strlen	"string.h"	strlen	Calculates the length of a string.
strncat	"string.h"	strncat	Adds a specified length of one string to another string.
strncmp	"string.h"	strncmp	Compares two strings up to a specified length.
strncpy	"string.h"	strncpy	Copies a specified

			length of one string into another.
strnicmp	"string.h"	strnicmp	Compares two strings up to a specified length, without sensitivity to case.
strnset	"string.h"	strnset - strset	Sets all characters in a specified length of string to a specified character.
strpbrk	"string.h"	strpbrk	Locates specified characters in a string.
strrchr	"string.h"	strrchr	Locates the last occurrence of a character within a string.
strrev	"string.h"	strrev	Reverses the order of characters in a string.
strset	"string.h"	strnset - strset	Sets all characters in a string to a specified character.
strspn	"string.h"	strspn	Locates the first character in a string that is not part of speci-

			fied set of characters.
strstr	"string.h"	strstr	Locates the first occurrence of a string in another string.
strtok	"string.h"	strtok	Locates a specified token in a string.
strxfrm	"string.h"	strxfrm	Transforms strings according to locale.

Character Testing

Function	Header File	Link to Function	Descriptio
"isalnum"	"ctype.h"	isalnum to isxdigit	Tests for alphanumeric characters.
"isalpha"	"ctype.h"	isalnum to isxdigit	Tests for alphabetic characters.
isascii	"ctype.h"	isascii	Tests if an integer is within the ASCII range.
"iscntrl"	"ctype.h"	isalnum to isxdigit	Tests for control characters.
_iscsym	"ctype.h"	_iscsym - _iscsymf	Tests if a character is

			alphabetic or an underscore.
_iscsymf	"ctype.h"	_iscsym - _iscsymf	Tests if a character is alphabetic, a digit, or an underscore.
"isdigit"	"ctype.h"	isalnum to isxdigit	Tests for decimal digits.
"isgraph"	"ctype.h"	isalnum to isxdigit	Tests for printable characters excluding the space.
"islower"	"ctype.h"	isalnum to isxdigit	Tests for lowercase letters.
"isprint"	"ctype.h"	isalnum to isxdigit	Tests for printable characters including the space.
"ispunct"	"ctype.h"	isalnum to isxdigit	Tests for printable characters excluding the space.
"isspace"	"ctype.h"	isalnum to isxdigit	Tests for whitespace characters.
"isupper"	"ctype.h"	isalnum to isxdigit	Tests for uppercase letters.
iswalnum	"wctype.h"	iswalnum to iswxdigit	Tests for alphanumeric wide characters.
iswalpha	"wctype.h"	iswalnum to iswxdigit	Tests for alpha-

			betic wide characters.
iswctype	"wctype.h"	iswctype	Tests a wide character for a specified property.
iswdigit	"wctype.h"	iswalnum to iswxdigit	Tests wide characters for decimal digits.
iswgraph	"wctype.h"	iswalnum to iswxdigit	Tests for printable wide characters excluding the space.
iswlower	"wctype.h"	iswalnum to iswxdigit	Tests wide characters for lowercase letters.
iswprint	"wctype.h"	iswalnum to iswxdigit	Tests for printable wide characters including the space.
iswpunct	"wctype.h"	iswalnum to iswxdigit	Tests for printable wide characters excluding the space.
iswspace	"wctype.h"	iswalnum to iswxdigit	Tests for white-space wide characters.
iswupper	"wctype.h"	iswalnum to iswxdigit	Tests wide characters for uppercase letters.

"isxdigit"	"ctype.h"	isalnum to isxdigit	Tests for hexadecima digits.
iswxdigit	"wctype.h"	iswalnum to iswxdigit	Tests wide characters for hexadecima digits.
wcwidth	"wchar.h"	wcwidth	Determines number of display positions required to display a given wide character.

Character Case Mapping

Function	Header File	Link to Function	Descriptio
strlwr	"string.h"	strlwr	Converts any uppercase letters in a string to lowercase.
strupr	"string.h"	strupr	Converts any lowercase letters in a string to uppercase.
_toascii	"ctype.h"	tolower - toupper	Converts a value to its ASCII character equivalent.
tolower	"ctype.h"	tolower - toupper	Converts

			a character to lowercase.
_tolower	"ctype.h"	_toascii - _tolower - _toupper	Converts an uppercase ASCII character to lowercase.
toupper	"ctype.h"	tolower - toupper	Converts a character to uppercase.
_toupper	"ctype.h"	_toascii - _tolower - _toupper	Converts a lowercase ASCII character to uppercase.
towlower	"wctype.h"	towlower - towupper	Converts a lowercase wide character to uppercase.
towupper	"wctype.h"	towlower - towupper	Converts a lowercase wide character to uppercase.

Wide Character String Operation Functions

Function	Header File	Link to Function	Descriptio
mblen	"stdlib.h"	mblen	Determines length of string.
wcscat	"wchar.h"	wcscat	Concatenates wchar_t strings.
wcschr	"wchar.h"	wcschr	Searches wchar_t

			string for character.
wcscmp	"wchar.h"	wcscmp	Compares wchar_t strings.
wcscoll	"wchar.h"	wcscoll	Compares two wide character strings based on collating elements for the current locale.
wcscpy	"wchar.h"	wcscpy	Copies wchar_t string.
wcscspn	"wchar.h"	wcscspn	Searches wchar_t string for characters.
wcslen	"wchar.h"	wcslen	Finds length of wchar_t string.
wcsncat	"wchar.h"	wcsncat	Concatenates wchar_t string segment.
wcsncmp	"wchar.h"	wcsncmp	Compares wchar_t string segments.
wcsncpy	"wchar.h"	wcsncpy	Copies wchar_t string segments.
wcspbrk	"wchar.h"	wcspbrk	Locates wchar_t characters in string.
wcsspn	"wchar.h"	wcsspn	Finds number of wchar_t characters.
wcsrchr	"wchar.h"	wcsrchr	Locates wchar_t character in

			string.
wcsstr	"wchar.h"	wcsstr	Locates the first occurrence of a wide character string within another wide character string.
wcstok	"wchar.h"	wcstok	Locates a specified token in a wide character string.
wcswcs	"wchar.h"	wcswcs	Locates a wchar_t string in another wchar_t string.
wcswidth	"wchar.h"	wcswidth	Determines the number of display positions required to display a given wide character string.
wcsxfrm	"wchar.h"	wcsxfrm	Transforms wide character strings according to the current locale.

Differentiating between Memory Management Functions

The memory management functions defined by ANSI are calloc, malloc, realloc, and free. These regular functions allocate and free memory from the default run-time heap. (*The Developer's Toolkit* has added another function, _heapmin, to return unused memory to the system.) *The Developer's Toolkit* also provides different versions of each of these functions as extensions to the ANSI definition.

All the versions actually work the same way; they differ only in what heap they allocate from, and in whether they save information to help you debug memory problems. The memory allocated by all of these functions is suitably aligned for storing any type of object.

The following table summarizes the different versions of memory management functions, using malloc as an example of how the names of the functions change for each version. They are all described in greater detail after the table.

Run-Time Heap	Regular Version
DEFAULT HEAP	malloc
USER HEAP	_umalloc

To use these extensions, you must set the language level to extended.

Heap-Specific Functions

Use the heap-specific versions to allocate and free memory from a user-created heap that you specify. (You can also explicitly use the run-time heap if you want.) Their names are prefixed by _u (for "user heaps"), for example, _umalloc, and they are defined in <umalloc.h>.

The functions provided are:

- _ucalloc
- _umalloc
- _uheapmin

Notice there is no heap-specific version of realloc or free. Because they both always check what heap the memory was allocated from, you can always use the regular versions regardless of what heap the memory came from.

For more information about creating your own heaps and using the heap-specific memory management functions, see "Managing Memory with Multiple Heaps" in the *VisualAge C++ Programming Guide* .

Infinity and NaN Support

The IBM Developer's Toolkit for OS/2 Warp Version 4 compiler supports the use of infinity and NaN (not-a-number) values. Infinity is a value with an associated sign that is mathematically greater in magnitude than any binary floating-point number. A NaN is a value in floating-point computations that is not interpreted as a mathematical value, and that contains a mask state and a sequence of binary digits.

The value of infinity can be computed from 1.0 / 0.0. The value of a NaN can be computed from 0.0 / 0.0.

Depending on its bit pattern, a NaN can be either quiet (NaNQ) or signaling (NaNS), as defined in the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic* (754-1982). A NaNQ is masked and never generates exceptions. A NaNS may be masked and may generate an exception, but does not necessarily do so. *The Developer's Toolkit for OS/2 Warp Version 4* compiler supports only quiet NaN values; all NaN values discussed below refer to quiet NaNs.

NaN and infinity values are defined as macro constants in the <float.h> header file. The macros are: compact break=fit.

Macro	**Description**
_INFINITYF	Infinity of type **float**
_INFINITY	Infinity of type **double**
_INFINITYL	Infinity of type **long double**
_INFF	Same as _INFINITYF
_INF	Same as _INFINITY
_INFL	Same as _INFINITYL

`_NANF`	Quiet NaN of type **float**
`_NAN`	Quiet NaN of type **double**
`_NANL`	Quiet NaN of type **long double**.

You can get the corresponding negative values by using the unary minus operator (for example, `-_INF`).

Note: The value of `0.0` can also be positive or negative. For example, `1.0 / (-0.0)` results in `-_INF`.

Because these macros are actually references to constant variables, you cannot use them to initialize static variables. For example, the following statements are not allowed:

```
   static double infval = _INF;
   static float nanval = 1.0 + _NANF;
```

However, you can initialize static variables to the numeric values of infinity and NaN:

```
   static double infval = 1.0 / 0.0;
   static float nanval =  0.0 / 0.0;
```

Note: Although positive and negative infinities are specific bit patterns, NaNs are not. A NaN value is not equal to itself or to any other value. For example, if you assign a NaN value to a variable `x`, you cannot check the value of `x` with the statement `if (_NAN == x)`. Instead, use the statement `if (x != x)`.

All relational and equality expressions involving NaN values always evaluate to FALSE or zero (`0`), with the exception of not equal (!=), which always evaluates to TRUE or one (`1`).

For information on the bit mapping and storage mapping of NaN and infinity values, see the *User's Guide* .

Infinity and NaN in Library Functions

When the language level is set to extended which is the default, infinity and NaN values can be passed as arguments to the `scanf` and `printf` families of library functions, and to the string conversion and math functions. At other language levels, these functions work as described in this book.

This section describes how the library functions handle the infinity and NaN values.

---

scanf Family

The `scanf` family of functions includes the functions `scanf`, `fscanf`, and `sscanf`. When reading in floating-point numbers, these functions convert the strings INFINITY, INF, and NAN (in uppercase, lowercase, or mixed case) to the corresponding floating-point value. The sign of the value is determined by the format specification.

Given a string that consists of NAN, INF, or INFINITY, followed by other characters, the scanf functions read in only the NaN or infinity value, and consider the rest of the string to be a second input field. For example, Nancy would be scanned as two fields, *Nan* and *cy* .

Note: In the case of a string that begins with INF, the functions check the fourth letter. If that letter is not `I` (in uppercase or lowercase), INF is read and converted and the rest of the string is left for the next format specification. If the fourth letter is `I`, the functions continue to scan for the full INFINITY string. If the string is other than INFINITY, the entire string is discarded.

Example: In the following example, `fscanf` converts NAN and INFINITY strings to their numeric values.

```
#include <stdio.h>

int main(void)
{
   int n, count;
   double d1, d2, d3;
   FILE *stream;

   stream = tmpfile();

   fputs(" INFINITY NAn INF", stream);

   rewind(stream);

   n = fscanf(stream, "%lF%lf%lF%n", &d1, &d2, &d3, &count);

   if (n != EOF)
   {
      printf("Number of fields converted = %d\n", n);
      printf("Number of characters read = %d\n", count);
      printf("Output = %f %F %F\n", d1, d2, d3);
   }

   return 0;

   /* The expected output is:

      Number of fields converted = 3
      Number of characters read = 17
      Output = infinity NAN INFINITY  */

}
```

For more information on the `scanf`, `fscanf`, and `sscanf` functions, see the entries for each function in Library Functions.

----

printf Family

The `printf` family of functions includes the functions `printf`, `fprintf`, `sprintf`, `vfprintf`, `vprintf`, and `vsprintf`. These functions convert floating-point values of INFINITY and NaN to the strings "INFINITY" or "infinity" and "NAN" or "nan".

The case is determined by the format specification, as is the sign (positive or negative). When converting these values, the `printf` functions ignore the precision width given by the format specification.

Example: In the following example, `printf` converts the NaN and INFINITY values and prints the corresponding string.

```
#include <stdio.h>
#include <float.h>

int main(void)
{
   double infval = -(_INF);
   float nanval = _NANF;

   printf("-_INF is the same as %-15.30f\n", infval);
   printf("_NANF is the same as %-15.30F\n", nanval);

   return 0;

   /* The expected output is:

      -_INF is the same as -infinity
      _NANF is the same as NAN        */

}
```

For more information on the `printf`, `fprintf`, `sprintf`, `vfprintf`, `vprintf`, and `vsprintf` functions, see the entries for each function in Library Functions.

String Conversion Functions

The string conversion functions that support infinity and NaN representations include the functions: atof, _atold, _ecvt, _fcvt, _gcvt, strtod, strtold, and wcstod.

The atof, _atold, strtod, strtold, and wcstod functions accept the strings INFINITY, INF, and NAN (in uppercase, lowercase, or mixed case) as input, and convert these strings to the corresponding macro value defined in <float.h>. The _ecvt, _fcvt, and _gcvt functions convert infinity and NaN values to the strings INFINITY and NAN, respectively.

Note: If a signaling NaN string is passed to a string conversion function, a quiet NaN value is returned, and no signal is raised.

Example: The following example uses atof to convert the strings "naN" and "inf" to the corresponding macro value.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *nanstr;
   char *infstr;

   nanstr = "naN";
   printf( "Result of atof = %.10e\n", atof(nanstr) );

   infstr = "inf";
   printf( "Result of atof = %.10E\n", atof(infstr) );

   return 0;

   /* The expected output is:

      Result of atof = nan
      Result of atof = INFINITY  */

}
```

For more information on the individual string conversion functions, refer to the entries for them in Library Functions.

Math Functions

Most math functions accept infinity, negative infinity, and NaN values as input. In general, a NaN value as input results in a NaN value as output, and infinity values as input usually result in infinity values. If the input value is outside the domain or range of the function, errno is set to EDOM or ERANGE, respectively.

The following tables display the results of each math function when NaN or infinity values are input, and the associated errno value if one exists. The first table lists the functions that take only one argument; the second lists those that take two arguments.

Note: In some cases, infinity is always a valid input value for the function regardless of the language level (for example, atan). These cases do not appear in these two tables.

Function	Input	Result	errno Value
acos	NaN	NaN	
asin	infinity	0	EDOM
	-infinity	0	EDOM
atan	NaN	NaN	
ceil	NaN	NaN	
floor	infinity	infinity	
	-infinity	-infinity	
cos	NaN	NaN	EDOM

tan	infinity	NaN	ERANGE
	-infinity	NaN	ERANGE
cosh	NaN	NaN	
	infinity	infinity	ERANGE
	-infinity	infinity	ERANGE
erf	NaN	NaN	EDOM
	infinity	1	
	-infinity	-1	
erfc	NaN	NaN	EDOM
	infinity	0	
	-infinity	2	
exp	NaN	NaN	
	infinity	infinity	ERANGE
	-infinity	0	ERANGE
fabs	NaN	NaN	
	infinity	infinity	
	-infinity	infinity	
frexp	NaN	NaN, 0	EDOM
	infinity	NaN, 0	EDOM
	-infinity	NaN, 0	EDOM
gamma	NaN	NaN	EDOM
	infinity	infinity	ERANGE
	-infinity	NaN	EDOM
log	NaN	NaN	
log10	infinity	infinity	
	0	-infinity	ERANGE
	<0	NaN	EDOM
modf	NaN	NaN, NaN	EDOM
	infinity	NaN, infinity	EDOM
	-infinity	NaN, -infinity	EDOM
sin	NaN	NaN	EDOM
	infinity	NaN	ERANGE
	-infinity	NaN	ERANGE
sinh	NaN	NaN	EDOM
	infinity	infinity	ERANGE
	-infinity	-infinity	ERANGE
sqrt	NaN	NaN	
	infinity	infinity	
	-infinity	0	EDOM
tanh	NaN	NaN	EDOM
	infinity	1	
	-infinity	-1	

The functions in the following table take two arguments. The results from NaN and infinity values vary depending on which argument they are passed as.

Function	Argument 1	Argument 2	Result	errno Value
atan2	NaN	any number	NaN	EDOM
	any number	NaN	NaN	

fmod	NaN	any number	NaN	EDOM
	any number	NaN	NaN	EDOM
	±infinity	any number	0	EDOM
ldexp	infinity	any number	infinity	ERANGE
	-infinity	any number	-infinity	ERANGE
	NaN	any number	NaN	EDOM
pow	±infinity	0	NaN	EDOM
	infinity	-infinity	NaN	EDOM
	-infinity	±infinity	NaN	EDOM
	-infinity	<-1	NaN	EDOM
	-infinity	<1, >-1	NaN	EDOM
	-infinity	>1	NaN	EDOM
	NaN	any number	NaN	EDOM
	any number	NaN	NaN	EDOM
	<=0	infinity	NaN	EDOM
	1	±infinity	NaN	EDOM
	±infinity	±1	0	ERANGE
	>0, <1	infinity	0	ERANGE
	>0, <1	-infinity	infinity	ERANGE

Note: If a signaling NaN is passed to a math function, the behavior is undefined.

Using Low-Level I/O Functions

The Developer's Toolkit compiler supports both stream and low-level I/O. The primary difference between the two types of I/O is that low-level I/O leaves the responsibility of buffering and formatting up to you.

In general, you should not mix input or output from low-level I/O with that from stream I/O. The only way to communicate between stream I/O and low-level I/O is by using the fdopen or fileno functions.

The low-level I/O functions include:

```
access
chmod
_chsize
close
creat
dup
dup2
__eof
fdopen
_filelength
fileno
fstat
isatty
lseek
open
read
_setmode
_sopen
stat
_tell
umask
write
```

When you use the low-level I/O functions, you should be aware of the following:

- A handle is a value that identifies a file. It is created by the system and used by low-level I/O functions. For *The Developer's* Toolkit, the handle returned by low-level I/O functions like open (called the *C_handle*) is the same as that returned by DosOpen (called the *API_handle*). As a result, you can get a file handle using the low-level I/O functions, and then use it with OS/2 APIs.

  Portability Note Other compilers may map the file handle so that the *C_handle* and *API_handle* are different. If you will be compiling your programs with other compilers, do not write code that depends on the file handles being the same.

  You can pass handles between library environments without restriction. If you acquire a handle other than by using *The Developer's Toolkit* library functions open, creat, _sopen, or fileno, you must run _setmode for that handle before you use it with other *The Developer's Toolkit* library functions.

- The default open-sharing mode is `SH_DENYWR`. Use _sopen to obtain other sharing modes.

- Text mode deletes `'\r'` characters on input and changes `'\n'` to `'\r\n'` on output.

- **In a multithread environment, you must ensure that two threads do not attempt to perform low-level I/O operations on the same file at the same time. You must make sure that one I/O process is completed before another begins**.

- If the file mode is text, the low-level I/O functions treat the character `'x1a'` in the following ways:
  - If it is detected in a nonseekable file, `'x1a'` is treated as end-of-file. In a seekable file, it is treated as end-of-file only if it is the last character.
  - If a file is opened as text with either the O_APPEND or O_RDWR flags and `'x1a'` is the last character of the file, the last character of the file is deleted.

Library Functions

The Developer's Toolkit for OS/2 Warp Version 4 libraries are divided into two parts:

- Standard libraries, which define the SAA features and *The Developer's Toolkit* standard extensions to SAA

- Subsystem libraries, which are a subset of the standard libraries, and are used for subsystem development. The functions in these libraries do not require a run-time environment.

This section lists alphabetically and describes all the functions that *The Developer's Toolkit for OS/2 Warp Version 4* product offers, including the extensions to the ANSI/ISO C definition. For information on the subsystem libraries and the functions in them, see the chapter called "Developing Subsystems" in the *VisualAge C++ Programming Guide* .

Each function description includes the following subsections:

Syntax
The prototyped declaration of the function and the header file in which it is found. To include the declaration in your code, include the header file.

Description
A brief description of what the function does, what parameters it takes, and how to use the function.

Returns
The value returned from a successful call to the function and the error return value.

Example Code
A short example of how to use the function. From the online C Library Reference, you can use the IPF **Copy to File** choice from the **Services** pull-down to copy a function example to a separate file (called TEXT.TMP by default). You can then compile, link, and run the example, or use the example code in your own source files.

Related Information
A list of other functions that are similar to or related to the function, and of other topics that provide additional information that help you use the function.

----

abort - Stop a Program

abort - Stop a Program

Syntax

```
#include <stdlib.h>
void abort(void);
```

Description

abort causes an abnormal program termination and returns control to the host environment. It is similar to `exit`, except that `abort` does not flush buffers and close open files before ending the program. Calls to `abort` raise the SIGABRT signal.

Returns

There is no return value.

Example Code

This example tests for successful opening of the file MYFILE.MJQ. If an error occurs, an error message is printed and the program ends with a call to `abort`.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *stream;

   if (NULL == (stream = fopen("myfile.mjq", "r"))) {
      perror("Could not open data file");
      abort();
   }
   return 0;

   /****************************************************************************
      If myfile.mjq does'nt exist, the output should be:

      Could not open data file: The file cannot be found.
   ****************************************************************************/
}
```

Related Information

- exit
- _exit
- signal

----

abs - Calculate Integer Absolute Value

abs - Calculate Integer Absolute Value

Syntax

```
#include <stdlib.h>
int abs(int n);
```

Description

abs returns the absolute value of an integer argument *n*.

Returns

There is no error return value. The result is undefined when the absolute value of the argument cannot be represented as an integer. The value of the minimum allowable integer is defined by `-INT_MAX` in the `<limits.h>` include file.

Example Code

This example calculates the absolute value of an integer `x` and assigns it to `y`.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
   int x = -4, y;

   y = abs(x);                                           /* y = 4            */
   printf("abs( %d ) = %d\n", x, y);
   return 0;

   /*****************************************************************************
      The output should be:

      abs( -4 ) = 4
   *****************************************************************************/
}
```

Related Information

- fabs
- labs

----

access - Determine Access Mode

access - Determine Access Mode

Syntax

```
#include <io.h>
int access(char *pathname, int mode);
```

Description

access determines whether the specified file exists and whether you can get access to it in the given *mode* . Possible values for the *mode* and their meaning in the `access` call are:

Value	**Meaning**
06	Check for permission to read from and write to the file.
04	Check for permission to read from the file.
02	Check for permission to write to the file.
00	Check only for the existence of the file.

Returns

access returns `0` if you can get access to the file in the specified *mode* . A return value of `-1` shows that the file does not exist or is inaccessible in the given *mode* , and the system sets `errno` to one of the following values:

Value	**Meaning**
EACCESS	Access is denied; the permission setting of the file does not allow you to get access to the file in the specified mode.
ENOENT	The system cannot find the file or the path that you specified, or the file name was incorrect.
EINVAL	The mode specified was not valid.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example checks for the existence of the file SAMPLE.DAT. If the file does not exist, it is created.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   if (-1 == access("sample.dat", 00)) {
      printf("File sample.dat does not exist.\n");
      printf("Creating sample.dat.\n");
      system("echo Sample Program > sample.dat");
      if (0 == access("sample.dat", 00))
         printf("File sample.dat created.\n");
   }
   else
      printf("File sample.dat exists.\n");
   return 0;

   /****************************************************************************
      The output should be:

      File sample.dat does not exist.
      Creating sample.dat.
      File sample.dat created.
   ****************************************************************************/
}
```

Related Information

- chmod
- _sopen
- umask

----

acos - Calculate Arccosine

acos - Calculate Arccosine

Syntax

```
#include <math.h>
double acos(double x);
```

Description

acos calculates the arccosine of *x*, expressed in radians, in the range `0` to the value of pi.

Returns

acos returns the arccosine of *x*. The value of *x* must be between -1 and 1 inclusive. If *x* is less than -1 or greater than 1, acos sets `errno` to `EDOM` and returns `0`.

Example Code

This example prompts for a value for *x*. It prints an error message if *x* is greater than 1 or less than -1; otherwise, it assigns the arccosine of *x* to *y*.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define  MAX           1.0
```

```
#define  MIN          -1.0

int main(void)
{
   double x,y;

   printf("Enter x\n");
   scanf("%lf", &x);

   /* Output error if not in range */
   if (x > MAX)
      printf("Error: %lf too large for acos\n", x);
   else
      if (x < MIN)
         printf("Error: %lf too small for acos\n", x);
      else {
         y = acos(x);
         printf("acos( %lf ) = %lf\n", x, y);
      }
   return 0;

   /****************************************************************************
      For the following input: 0.4

      The output should be:

      Enter x
      acos( 0.400000 ) = 1.159279
   ****************************************************************************/
}
```

Related Information

- asin
- atan - atan2
- cos
- cosh
- sin
- sinh
- tan
- tanh

----

asctime - Convert Time to Character String

asctime - Convert Time to Character String

Syntax

```
#include <time.h>
char *asctime(const struct tm *time);
```

Description

The `asctime` function converts time, stored as a structure pointed to by *time* , to a character string. You can obtain the *time* value from a call to `gmtime` or `localtime`; either returns a pointer to a `tm` structure defined in `<time.h>`. See gmtime for a description of the `tm` structure fields.

The string result that `asctime` produces contains exactly 26 characters and has the format:

```
   "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

See printf for a description of format specifications. The following are examples of the string returned:

```
   Sat Jul 14 02:03:55 1995\n\0
```

or

```
   Sat Jul 14  2:03:55 1995\n\0
```

The `asctime` function uses a 24-hour-clock format. The days are abbreviated to: `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, and `Sat`. The months are abbreviated to: `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, and `Dec`. All fields have constant width. Dates with only one digit are preceded either with a zero or a blank space. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Returns

The `asctime` function returns a pointer to the resulting character string. There is no error return value.

Note: `asctime`, `ctime`, and other time functions may use a common, statically allocated buffer to hold the return string. Each call to one of these functions may destroy the result of the previous call.

Example Code

This example polls the system clock and prints a message giving the current time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   struct tm *newtime;
   time_t ltime;

   /* Get the time in seconds */
   time(&ltime);

   /* Convert it to the structure tm */
   newtime = localtime(&ltime);

   /* Print the local time as a string */
   printf("The current date and time are %s", asctime(newtime));
   return 0;

   /****************************************************************************
      The output should be similar to :

      The current date and time are Fri Jun 28 13:51 1995
   ****************************************************************************/
}
```

Related Information

- ctime
- gmtime
- localtime
- mktime
- strftime
- time
- printf

----

asin - Calculate Arcsine

asin - Calculate Arcsine

Syntax

```
#include <math.h>
double asin(double x);
```

Description

asin calculates the arcsine of *x*, in the range -pi/2 to pi/2 radians.

Returns

asin returns the arcsine of *x*. The value of *x* must be between -1 and 1. If *x* is less than -1 or greater than 1, `asin` sets `errno` to `EDOM`, and returns a value of `0`.

Example Code

This example prompts for a value for *x*. It prints an error message if *x* is greater than 1 or less than -1; otherwise, it assigns the arcsine of *x* to *y*.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define  MAX           1.0
#define  MIN          -1.0

int main(void)
{
   double x,y;

   printf("Enter x\n");
   scanf("%lf", &x);

   /* Output error if not in range                                            */
   if (x > MAX)
      printf("Error: %lf too large for asin\n", x);
   else
      if (x < MIN)
         printf("Error: %lf too small for asin\n", x);
      else {
         y = asin(x);
         printf("asin( %lf ) = %lf\n", x, y);
      }
   return 0;

   /****************************************************************************
      For the following input: 0.2

      The output should be:

      Enter x
      asin( 0.200000 ) = 0.201358
   ****************************************************************************/
}
```

Related Information

- acos
- atan - atan2
- cos
- cosh
- sin
- sinh
- tan
- tanh

----

assert - Verify Condition

assert - Verify Condition

Syntax

```
#include <assert.h>
void assert(int expression);
```

Description

assert prints a diagnostic message to stderr and aborts the program if *expression* is false (zero). The diagnostic message has the format:

```
Assertion failed: expression, file filename, line line-number.
```

assert takes no action if the *expression* is true (nonzero).

Use `assert` to identify program logic errors. Choose an *expression* that holds true only if the program is operating as you intend. After you have debugged the program, you can use the special no-debug identifier `NDEBUG` to remove the `assert` calls from the program. If you define `NDEBUG` to any value with a **#define** directive, the C preprocessor expands all assert invocations to **void** expressions. If you use `NDEBUG`, you must define it before you include `<assert.h>` in the program.

Returns

There is no return value.

Note: assert is implemented as a macro. Do not use the **#undef** directive with assert.

Example Code

In this example, assert tests *string* for a null string and an empty string, and verifies that *length* is positive before processing these arguments.

```
#include <stdio.h>
#include <assert.h>

void analyze(char *string,int length)
{
   assert(string != NULL);                           /* cannot be NULL   */
   assert(*string != '\0');                          /* cannot be empty  */
   assert(length > 0);                               /* must be positive */
   return;
}

int main(void)
{
   char *string = "ABC";
   int length = 3;

   analyze(string, length);
   printf("The string %s is not null or empty, and has length %d \n", string,
      length);
   return 0;

   /*****************************************************************************
      The output should be:

      The string ABC is not null or empty, and has length 3
   ****************************************************************************/
}
```

Related Information

- abort
- `#define` in the *Language Reference*
- `#undef` in the *Language Reference*

----

atan - atan2 - Calculate Arctangent

atan - atan2 - Calculate Arctangent

Syntax

```
#include <math.h>
double atan(double x);
double atan2(double y, double x);
```

Description

atan and atan2 calculate the arctangent of *x* and *y/x*, respectively.

Returns

atan returns a value in the range -pi/2 to pi/2 radians. atan2 returns a value in the range -pi to pi radians. If both arguments of `atan2` are zero, the function sets `errno` to `EDOM`, and returns a value of `0`.

Example Code

This example calculates arctangents using the `atan` and `atan2` functions.

```
#include <math.h>

int main(void)
{
   double a,b,c,d;

   c = 0.45;
   d = 0.23;
   a = atan(c);
   b = atan2(c, d);
   printf("atan( %lf ) = %lf\n", c, a);
   printf("atan2( %lf, %lf ) = %lf\n", c, d, b);
   return 0;

   /****************************************************************************
      The output should be:

      atan( 0.450000 ) = 0.422854
      atan2( 0.450000, 0.230000 ) = 1.098299
   ****************************************************************************/
}
```

Related Information

- acos
- asin
- cos
- cosh

- sin
- sinh
- tan
- tanh

----

atexit - Record Program Termination Function

atexit - Record Program Termination Function

Syntax

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

`atexit` records a function, pointed to by *func* , that the system calls at normal program termination. For portability, you should use `atexit` to register up to 32 functions only. The functions are executed in a LIFO (last-in-first-out) order.

Returns

`atexit` returns `0` if it is successful, and nonzero if it fails.

Example Code

This example uses atexit to call the function `goodbye` at program termination.

```
#include <stdlib.h>
#include <stdio.h>

void goodbye(void)
{

   /* This function is called at normal program termination                   */

   printf("The function goodbye was called at program termination\n");
}

int main(void)
{
   int rc;

   rc = atexit(goodbye);
   if (rc != 0)
      perror("Error in atexit");
   return 0;

   /****************************************************************************
      The output should be:

      The function goodbye was called at program termination
   ****************************************************************************/
}
```

Related Information

- exit
- _exit
- _onexit
- signal

----

atof - Convert Character String to Float

atof - Convert Character String to Float

Syntax

```
#include <stdlib.h>
double atof(const char *string);
```

Description

atof converts a character string to a double-precision floating-point value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

atof expects a *string* in the following form:

```
>>                                                        >
      whitespace      +    digits
                                       .    digits
                             . digits

>                            ><
      e           digits
      E     +
```

The actual decimal point character (radix character) is determined by the LC_NUMERIC category of the current locale.

The white space consists of the same characters for which the `isspace` function is true, such as spaces and tabs. atof ignores leading white-space characters.

For atof, *digits* is one or more decimal digits; if no digits appear before the decimal point, at least one digit must appear after the decimal point. The decimal digits can precede an exponent, introduced by the letter e or E. The exponent is a decimal integer, which may be signed.

`atof` will not fail if a character other than a digit follows an `E` or if `e` is read in as an exponent. For example, `100elf` will be converted to the floating-point value `100.0`. The accuracy is up to 17 significant character digits. The *string* can also be `"infinity"`, `"inf"`, or `"nan"`. These strings are case-insensitive, and can be preceded by a unary minus (`-`). They are converted to infinity and NaN values.

Returns

atof returns a `double` value produced by interpreting the input characters as a number. The return value is `0` if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

Example Code

This example shows how to convert numbers stored as strings to numerical values using the `atof` function.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
   double x;
   char *s;

   s = " -2309.12E-15";
   x = atof(s);                                           /* x = -2309.12E-15 */
   printf("atof( %s ) = %G\n", s, x);
   return 0;

   /****************************************************************************
      The output should be:

      atof(  -2309.12E-15 ) = -2.30912E-12
   ****************************************************************************/
}
```

Related Information

- atoi
- atol
- _atold
- strtod
- strtol
- strtold
- strtoul
- Infinity and NaN Support

----

atoi - Convert Character String to Integer

atoi - Convert Character String to Integer

Syntax

```
#include <stdlib.h>
int atoi(const char *string);
```

Description

atoi converts a character string to an integer value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

atoi does not recognize decimal points nor exponents. The `string` argument for this function has the form:

```
 >>                         digits  ><
       whitespace      +
```

where *whitespace* consists of the same characters for which the `isspace` function is true, such as spaces and tabs. atoi ignores leading white-space characters. *digits* is one or more decimal digits.

Returns

`atoi` returns an `int` value produced by interpreting the input characters as a number. The return value is `0` if the function cannot convert the input to a value of that type. The return value is undefined in the case of an overflow.

Example Code

This example shows how to convert numbers stored as strings to numerical values using the `atoi` function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int i;
   char *s;

   s = " -9885";
   i = atoi(s);                                           /* i = -9885       */
   printf("atoi( %s ) = %d\n", s, i);
   return 0;

   /*****************************************************************************
      The output should be:

      atoi(  -9885 ) = -9885
   *****************************************************************************/
}
```

Related Information

- atof
- atol
- _atold
- strtod
- strtol
- strtold

----

atol - Convert Character String to Long Integer

atol - Convert Character String to Long Integer

Syntax

```
#include <stdlib.h>
long int atol(const char *string);
```

Description

atol converts a character string to a long value.

The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number; this character can be the null character that ends the string.

`atol` does not recognize decimal points nor exponents. The *string* argument for this function has the form:

```
>>                           digits  ><
      whitespace      +
```

where *whitespace* consists of the same characters for which the `isspace` function is true, such as spaces and tabs. atol ignores leading white-space characters. *digits* is one or more decimal digits.

Returns

`atol` returns a `long` value produced by interpreting the input characters as a number. The return value is `0L` if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

Example Code

This example shows how to convert numbers stored as strings to numerical values using the `atol` function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long l;
   char *s;

   s = "98854 dollars";
   l = atol(s);                                           /* l = 98854        */
   printf("atol( %s ) = %d\n", s, l);
   return 0;

   /*****************************************************************************
      The output should be similar to :

      atol( 98854 dollars ) = 98854
   ****************************************************************************/
}
```

Related Information

- atof
- atoi
- _atold
- strtod
- strtol
- strtold

_atold - Convert Character String to Long Double

_atold - Convert Character String to Long Double

Syntax

```
#include <stdlib.h>    /* also defined in <math.h> */
long double _atold(const char *nptr);
```

Description

_atold converts a character string pointed to by *nptr* to a `long double` value. The function continues until it reads a character it does not recognize as part of a number. This character can be the ending null character. Except for its behavior on error, _atold is equivalent to:

```
strtold(nptr, (char **)NULL)
```

The string pointed to by *nptr* must have the following format:

```
>>                                                               >
      whitespace                digits
                         +                 .    digits
                                .  digits

>                                 ><
       e              digits
       E       +
```

digits is one or more decimal digits. If no digits appear before the decimal point, at least one digit must follow the decimal point. The decimal point character (radix character) is determined by the LC_NUMERIC category of the current locale. You can place an exponent expressed as a decimal integer after the digits. The exponent can be signed.

The value of *nptr* can also be one of the strings `infinity`, `inf`, or `nan`. These strings are case-insensitive, and can be preceded by a unary minus (`-`). They are converted to infinity and NaN values. For more information on NaN and infinity values, see Infinity and NaN Support.

_atold ignores any white-space characters, as defined by the `isspace` function.

Returns

_atold returns the converted long double value. In the case of an underflow, it returns `0`. In the case of a positive overflow, _atold returns positive `_LHUGE_VAL`. It returns negative `_LHUGE_VAL` for a negative overflow.

Example Code

This example uses _atold to convert two strings, `"  -001234.5678e10end of string"` and `"NaNQ"`, to their corresponding long double values.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string;

   string = "  -001234.5678e10end of string";
   printf("_atold = %.10Le\n", _atold(string));
   string = "NaNQ";
   printf("_atold = %.10Le\n", _atold(string));
   return 0;

   /****************************************************************************
      The output should be:

      _atold = -1.2345678000e+13
      _atold = nan
   ****************************************************************************/

}
```

Related Information

- atof
- atoi
- atol
- strtod
- strtol
- strtold

_beginthread - Create New Thread

_beginthread - Create New Thread

Syntax

```
#include <stdlib.h>   /* also in <process.h> */
int _beginthread(void (*start_address) (void *),
                     (void *)stack,
                     unsigned stack_size,
                     void *arglist);
```

Description

_beginthread creates a new thread. It takes the following arguments:

start_address This parameter is the address of the function that the newly created thread will execute. When the thread returns from that function, it is terminated automatically. You can also explicitly terminate the thread by calling _endthread.

stack This parameter is ignored, but is retained to ease migration of C/2 programs. The C/2 compiler requires the second parameter to be the address of the bottom of the stack that the new thread will use. Because the OS/2 operating system automatically takes care of stack allocation, the parameter is not needed.

stack_size The size of the stack, in bytes, that is to be allocated for the new thread. The stack size should be a nonzero multiple of 4K and a minimum of 8K. Memory is used when needed, one page at a time.

arglist A parameter to be passed to the newly created thread. It is the size of a pointer, and is usually the address of a data item to be passed to the new thread, such as a `char` string. It provides _beginthread with a value to pass to the child thread. `NULL` can be used as a placeholder.

An alternative to this function is the OS/2 DosCreateThread function. If you use DosCreateThread, you must also use a `#pragma` handler statement for the thread function to ensure proper C exception handling. You should also call the _fpreset function at the start of the thread to preset the 387 control status word correctly. Using DosCreateThread requires that you use _endthread to terminate the thread.

Returns

If successful, _beginthread returns the thread ID number of the new thread. It returns -1 to indicate an error.

Example Code

This example uses _beginthread to start a new thread `bonjour`, which prints `Bonjour!` five times and then implicitly ends itself. The program then prints a statement indicating the thread identifier number for `bonjour`.

```
#define  INCL_DOS
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>

static int wait = 1;

void bonjour(void *arg)
{
   int i = 0;

   while (wait)             /* wait until the thread id has been printed     */
      DosSleep(0l);
   while (i++ < 5)
      printf("Bonjour!\n");
}

int main(void)
{
   int tid;
   tid = _beginthread(bonjour, NULL, 8192, NULL);
   if (-1 == tid) {
      printf("Unable to start thread.\n");
      return EXIT_FAILURE;
   }
   else {
      printf("Thread started with thread identifier number %d.\n", tid);
      wait = 0;
   }
   DosWaitThread((PTID)&tid, DCWW_WAIT);  /* wait for thread bonjour to       */
                                          /* end before ending main thread    */

   return 0;

   /****************************************************************************
      The output should be similar to :

       Thread started with thread identifier number 2.
       Bonjour!
       Bonjour!
       Bonjour!
       Bonjour!
       Bonjour!
   ****************************************************************************/
}
```

Related Information

- _endthread
- _threadstore

Bessel Functions - Solve Differential Equations

Bessel Functions - Solve Differential Equations

Syntax

```
#include <math.h>
double _j0(double x);
double _j1(double x);
double _jn(int n, double x);
double _y0(double x);
double _y1(double x);
double _yn(int n, double x);
```

Description

Bessel functions solve certain types of differential equations. The _j0, _j1, and _jn functions are Bessel functions of the first kind for orders 0, 1, and *n*, respectively.

The _y0, _y1, and _yn functions are Bessel functions of the second kind for orders 0, 1, and *n*, respectively. The argument *x* must be positive. The argument *n* should be greater than or equal to zero. If *n* is less than zero, it will be a negative exponent.

Returns

For _j0, _j1, _y0, or _y1, if the absolute value of *x* is too large, the function sets errno to ERANGE, and returns 0. For _y0, _y1, or _yn, if *x* is negative, the function sets errno to EDOM and returns the value -HUGE_VAL. For _y0, _y1, or _yn, if *x* causes an overflow, the function sets errno to ERANGE and returns the value -HUGE_VAL.

Example Code

This example computes y to be the order 0 Bessel function of the first kind for x, and z to be the order 3 Bessel function of the second kind for x.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double x,y,z;

   x = 4.27;
   y = j0(x);      /* y = -0.3660 is the order 0 bessel                       */
                   /* function of the first kind for x                        */

   printf("j0( 4.27 ) = %5.4f\n", y);
   z = yn(3, x);   /* z = -0.0875 is the order 3 bessel                       */
                   /* function of the second kind for x                       */

   printf("yn( 3,4.27 ) = %5.4f\n", z);
   return 0;

   /****************************************************************************
      The output should be:

      j0( 4.27 ) = -0.3660
      yn( 3,4.27 ) = -0.0875
   ****************************************************************************/
}
```

Related Information

- erf - erfc
- gamma

----

bsearch - Search Arrays

bsearch - Search Arrays

Syntax

```
#include <stdlib.h>  /* also in <search.h> */
void *bsearch(const void *key, const void *base,
              size_t num, size_t size,
              int (*compare)(const void *key, const void *element));
```

Description

bsearch performs a binary search of an array of *num* elements, each of *size* bytes. The array must be sorted in ascending order by the function pointed to by *compare*. The *base* is a pointer to the base of the array to search, and *key* is the value being sought.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. bsearch calls this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values: compact break=fit.

Value	Meaning
Less than 0	*key* less than *element*
0	*key* identical to *element*
Greater than 0	*key* greater than *element*

Returns

bsearch returns a pointer to *key* in the array to which *base* points. If two keys are equal, the element that *key* will point to is unspecified. If `bsearch` cannot find the *key*, it returns `NULL`.

Example Code

This example performs a binary search on the `argv` array of pointers to the program parameters and finds the position of the argument `PATH`. It first removes the program name from `argv`, and then sorts the array alphabetically before calling `bsearch`. The functions `compare1` and `compare2` compare the values pointed to by `arg1` and `arg2` and return the result to `bsearch`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int compare1(const void *arg1,const void *arg2)
{
   return (strcmp(*(char **)arg1, *(char **)arg2));
}

int compare2(const void *arg1,const void *arg2)
{
   return (strcmp((char *)arg1, *(char **)arg2));
}

int main(int argc,char *argv[])
{
   char **result;
   char *key = "PATH";
   int i;
   argv++;
   argc--;

   qsort((char *)argv, argc, sizeof(char *), compare1);
   result = (char **)bsearch(key, argv, argc, sizeof(char *), compare2);
   if (result != NULL) {
      printf("result = <%s>\n", *result);
   }
   else
      printf("result is null\n");
   return 0;

   /****************************************************************************
      If the program name is progname and you enter:

      progname where is PATH in this phrase?

      The output should be:

      result = <PATH>
   ****************************************************************************/
}
```

Related Information

- lfind - lsearch
- qsort

----

calloc - Reserve and Initialize Storage

calloc - Reserve and Initialize Storage

Syntax

```
#include <stdlib.h>  /* also in <malloc.h> */
void *calloc(size_t num, size_t size);
```

Description

calloc reserves storage space for an array of *num* elements, each of length *size* bytes. calloc then gives all the bits of each element an initial value of `0`.

Heap-specific versions of this function (_ucalloc) are also available. calloc always allocates memory from the default heap. You can also use the debug version of calloc, _debug_calloc, to debug memory problems.

Returns

calloc returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is `NULL` if there is not enough storage, or if *num* or *size* is `0`.

Example Code

This example prompts for the number of array entries required and then reserves enough space in storage for the entries. If `calloc` is successful, the example prints out each entry; otherwise, it prints out an error.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long *array;                            /* start of the array             */
   long *index;                            /* index variable                 */
   int i;                                  /* index variable                 */
   int num;                                /* number of entries of the array */

   printf("Enter the size of the array\n");
   scanf("%i", &num);

   /* allocate num entries                                                   */

   if ((index = array = calloc(num, sizeof(long))) != NULL) {
      for (i = 0; i < num; ++i)            /* put values in array            */
         *index++ = i;                     /* using pointer notation         */
      for (i = 0; i < num; ++i)            /* print the array out            */
         printf("array[ %i ] = %i\n", i, array[i]);
   }
   else {                                                /* out of storage   */
      perror("Out of storage");
      abort();
   }
   return 0;

   /****************************************************************************
      The output should be similar to :

      Enter the size of the array
      3
      array[ 0 ] = 0
      array[ 1 ] = 1
      array[ 2 ] = 2
   ****************************************************************************/
}
```

Related Information

- free
- malloc
- realloc
- _ucalloc
- "Managing Memory" in the *VisualAge C++ Programming Guide*

----

catclose - Closes a Specified Message Catalog

catclose - Closes a Specified Message Catalog

Syntax

```
#include <nl_types.h>

int catclose(nl_catd catd);
```

Description

The catclose function is used to close the catalog message file specified by the parameter catd that was previously opened by catopen.

Returns

- Successful return is 0.
- Unsuccessful return is -1.
- ERRNO is set.

Example Code

This example opens and closes a catalog message file.

```
#include <stdio.h>
#include <nl_types.h>

void load_cat(char *tcat)
{
  nl_catd catd;

  if ((catd = catopen(tcat, 0)) == CATD_ERR)
  {
     printf("Unable to load specified catalog. \n");
     exit (1);
  }

  if (catclose(catd) == -1)
     printf("Error when trying to close catalog file");

}
```

Related Information

- catopen
- catgets

----

catgets - Retrieves a Message from a Catalog

catgets - Retrieves a Message from a Catalog

Syntax

```
#include <nl_types.h>
```

```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

Description

The catgets subroutine retrieves a message from a catalog and constructs a message based on the set_id + msg_id + message. If the catgets subroutine finds the specified message, it loads it into an internal character string buffer, ends the message string with a null character, and returns a pointer to the buffer.

The returned pointer is used to reference the buffer and display the message. However, the buffer cannot be referenced after the catalog is closed.

The parameters are:

catd Specifies the open catalog to use for message retrieval.

set_id Specifies the set ID of the message.

msg_id Specifies the message ID of the message. The set_id and msg_id parameters specify a particular message in the catalog to retrieve.

s Specifies the default character-string buffer to use if the message is not retrieved from the catalog.

Returns

If the catgets subroutine is unsuccessful for any reason, it returns the user-supplied default message string specified by the **s** parameter.

Example Code

This example opens a message file with the name contained in tcat and prints out the message associated with set number in setno and the message number in msgno.

```
#include <stdio.h>
#include <nl_types.h>

char *catg(char *tcat, int setno, int msgno, char *def)
{
    nl_catd catd;                       /* Catalog descriptor. */

    if ((catd = catopen(tcat, 0)) == CATD_ERR)
    {
       printf("Unable to load specified catalog. \n");
       exit (1);
    }

    printf("ERROR MESSAGE : %s\n",
           catgets(catd, setno, msgno, def));

    if (catclose(catd) == -1)
       perror("Error when trying to close catalog file");

}
```

Related Information

- catopen
- catclose

----

catopen - Opens a Specified Message Catalog

catopen - Opens a Specified Message Catalog

Syntax

```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);
```

Description

The catopen subroutine opens a specified message catalog and returns a catalog descriptor used to retrieve messages from the catalog. The contents of the catalog descriptor are complete when the catgets function accesses the message catalog. The **nl_catd** data type is used for catalog descriptors. This data type is defined in the **nl_types.h** file.

If the catalog file name referred to by the *CatalogName* parameter contains a drive letter or a \, it is assumed to be an absolute pathname. That is, the catalog looked for following that path. If the catalog file name is not an absolute path name, the user environment determines which directory paths to search. The *NLSPATH* environment variable defines the directory search path.

Single-letter keywords for the *NLSPATH* are used as special variables as follows:

`%N`	The value of the *name* parameter that is passed to catopen.
`%L`	The value of the category **LC_MESSAGES**.
`%l`	The language element of the category **LC_MESSAGES**.
`%t`	The territory element of the category **LC_MESSAGES**.
`%c`	The codeset element of the category **LC_MESSAGES**.
`%%`	A single '%' character.

The value of the **LC_MESSAGES** category can be set by specifying values for the **LANG**, **LC_ALL**, or **LC_MESSAGES** environment variable. The value of the **LC_MESSAGES category** indicates which locale specific directory to search for message catalogs. For example, if the **catopen** subroutine specifies a catalog with the name mycmd, and the environment variables are set as follows:

```
NLSPATH=..\%N;\%N;\system\nls\%L\%N;\system\nls\%N
LANG=Fr_FR.IBM-850
```

then the application searches for the catalog in the following order:

- ..\mycmd
- .\mycmd
- \system\nls\Fr_FR.IBM-850\mycmd
- \system\nls\mycmd

If you omit the **%N** variable in a directory specification within the **NLSPATH** environment variable, the application assumes that the path defines a directory and searches for the catalog in that directory before searching the next specified path.

If the **NLSPATH** environment variable is not defined, the default path of . (current directory) is used.

The parameters are:

name
Specifies the catalog file to open.

oflag
If the value of the *oflag* argument is 0, the LANG environment variable is used to locate the catalog without regard to the LC_MESSAGES category. If the *oflag* argument is **NL_CAT_LOCALE**, the LC_MESSAGES category is used to locate the message catalog.

Returns

The catopen subroutine returns a catalog descriptor.

If the catopen subroutine returns a value of **CATD_ERR** ((nl_catd) -1), an error has occurred during creation of the **nl_catd** structure.

Example Code

```
#include <stdio.h>
#include <nl_types.h>

void load_cat(char *tcat)
{
  nl_catd catd;                         /* Catalog descriptor. */

  if ((catd = catopen(tcat, 0)) == CATD_ERR)
  {
     printf("Unable to load specified catalog. \n");
     exit(1);
  }

  if (catclose(catd) == -1)
     printf("Error when trying to close catalog file\n");

}
```

Related Information

- setlocale
- catgets
- catclose

----

ceil - Find Integer >= Argument

ceil - Find Integer >= Argument

Syntax

```
#include <math.h>
double ceil(double x);
```

Description

ceil computes the smallest integer that is greater than or equal to *x*.

Returns

ceil returns the integer as a `double` value.

Example Code

This example sets *y* to the smallest integer greater than 1.05, and then to the smallest integer greater than -1.05. The results are 2.0 and -1.0, respectively.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double y,z;

   y = ceil(1.05);                                        /* y = 2.0          */
   printf("ceil( 1.05 ) = %5.f\n", y);
   z = ceil(-1.05);                                       /* z = -1.0         */
   printf("ceil( -1.05 ) = %5.f\n", z);
   return 0;

   /***************************************************************************
      The output should be:
```

```
      ceil( 1.05 ) =     2
      ceil( -1.05 ) =    -1
   ****************************************************************************/
}
```

Related Information

- floor
- fmod

_cgets - Read String of Characters from Keyboard

_cgets - Read String of Characters from Keyboard

Syntax

```
#include <conio.h>
char *_cgets(char *str);
```

Description

_cgets reads a string of characters directly from the keyboard and stores the string and its length in the location pointed to by *str*.

_cgets continues to read characters until it meets a carriage return followed by a line feed (CR-LF) or until it reads the specified number of characters. It stores the string starting at *str* [2]. If _cgets reads a CR-LF combination, it replaces this combination with a null character ('\0') before storing the string. _cgets then stores the actual length of the string in the second array element, *str* [1].

The *str* variable must be a pointer to a character array. The first element of the array, *str* [0], must contain the maximum length, in characters, of the string to be read. The array must have enough elements to hold the string, a final null character, and 2 additional bytes.

Returns

If successful, _cgets returns a pointer to the actual start of the string, *str* [2]. Otherwise, _cgets returns NULL.

Example Code

This example creates a buffer and initializes the first byte to the size of the buffer. The program then accepts an input string using _cgets and displays the size and text of that string.

```
#include <conio.h>
#include <stdio.h>

void nothing(void)
{
}

int main(void)
{
   char buffer[82] =  { 84,0 };
   char *buffer2;
   int i;

   _cputs("\nPress any key to continue.");
   printf("\n");
   while (0 == _kbhit()) {
      nothing();
   }
   _getch();
   _cputs("\nEnter a line of text:");
   printf("\n");
   buffer2 = _cgets(buffer);
   printf("\nText entered was: %s", buffer2);
   return 0;

   /****************************************************************************
      The output should be similar to:

      Press any key to continue.

      Enter a line of text:
      This is a simple test.
      Text entered was: This is a simple test.

   ****************************************************************************/
}
```

Related Information

- _cputs
- fgets
- gets
- _getch - _getche

----

chdir - Change Current Working Directory

chdir - Change Current Working Directory

Syntax

```
#include <direct.h>
int chdir(char *pathname);
```

Description

chdir causes the current working directory to change to the directory specified by *pathname* . The *pathname* must refer to an existing directory.

Note: This function can change the current working directory on any drive. It cannot change the default drive. For example, if `A:\BIN` is the current working directory and `A:` is the default drive, the following changes only the current working directory on drive `C:`.

```
   chdir ("c:\\emp");
```

`A:` is still the default drive.

An alternative to this function is the DosSetCurrentDir call.

Returns

chdir returns a value of `0` if the working directory was successfully changed. A return value of -1 indicates an error; chdir sets errno to ENOENT, showing that chdir cannot find the specified path name. No error occurs if *pathname* specifies the current working directory.

Example Code

This example changes the current working directory to the root directory, and then to the `\red\green\blue` directory.

```
#include <direct.h>
#include <stdio.h>

int main(void)
{
   printf("Changing to the root directory.\n");
   if (chdir("\\"))
      perror(NULL);
   else
      printf("Changed to the root directory.\n\n");
   printf("Changing to directory '\\red\\green\\blue'.\n");
   if (chdir("\\red\\green\\blue"))
      perror(NULL);
   else
      printf("Changed to directory '\\red\\green\\blue'.\n");
   return 0;

   /****************************************************************************
      If directory \red\green\blue exists, the output should be:

      Changing to the root directory.
      Changed to the root directory.

      Changing to directory '\red\green\blue'.
      Changed to directory '\red\green\blue'.
   ****************************************************************************/
}
```

Related Information

- _chdrive
- _getcwd
- _getdcwd
- _getdrive
- system
- mkdir
- rmdir
- system

_chdrive - Change Current Working Drive

_chdrive - Change Current Working Drive

Syntax

```
#include <direct.h>
int _chdrive(int drive);
```

Description

_chdrive changes the current working drive to the *drive* specified. The *drive* variable is an integer value representing the number of the new working drive (A: is 1, B: is 2, and so on).

To change the default drive, include <os2.h>, define INCL_DOSFILEMGR, and use DosSetDefaultDisk to pass the appropriate command to the operating system. You can also use DosQueryCurrentDisk to query the disk. For more information, refer to the *Control Program Guide and Reference* .

Returns

_chdrive returns 0 if it is successful in changing the working drive. A return value of -1 indicates an error; _chdrive sets errno to EOS2ERR.

Example Code

This example uses _chdrive to change the current working drive to C:.

```
#include <direct.h>
#include <stdio.h>

int main(void)
{
   if (_chdrive(3))
      printf("Cannot change current working drive to 'C' drive.\n");
   else {
      printf("Current working drive changed to ");
      printf("'%c' drive.\n", ('A'+_getdrive()-1));
   }
   return 0;

   /****************************************************************************
      The output should be similar to :

      Current working drive changed to 'C' drive.
   ****************************************************************************/
}
```

Related Information

- chdir
- _getcwd
- _getdcwd
- _getdrive
- mkdir
- rmdir

----

chmod - Change File Permission Setting

chmod - Change File Permission Setting

Syntax

```
#include <io.h>
#include <sys\stat.h>
int chmod(char *pathname, int pmode);
```

Description

chmod changes the permission setting of the file specified by *pathname* . The permission setting controls access to the file for reading or writing. You can use chmod only if the file is closed.

The *pmode* expression contains one or both of the constants S_IWRITE and S_IREAD, defined in <sys\stat.h>. The meanings of the values of *pmode* are:

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If you do not give permission to write to the file, chmod makes the file read-only. With the OS/2 operating system, all files are readable; you cannot give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE set the same permission.

Specifying a *pmode* of S_IREAD is similar to making a file read-only with the ATTRIB system command.

Returns

chmod returns the value `0` if it successfully changes the permission setting. A return value of `-1` shows an error; chmod sets `errno` to one of the following values:

Value	**Meaning**
ENOENT	The system cannot find the file or the path that you specified, or the file name was incorrect.
EOS2ERR	The call to the operating system was not successful.
EINVAL	The mode specified was not valid.

Example Code

This example opens the file `chmod.dat` for writing after checking the file to see if writing is permissible. It then writes from the buffer to the opened file. This program takes file names passed as arguments and sets each to read-only.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   if (-1 == access("chmod.dat", 00))        /* Check if file exists.         */
      {
      printf("\nCreating chmod.dat.\n");
      system("echo Sample Program > chmod.dat");
      printf("chmod chmod.dat to be readonly.\n");
      if (-1 == chmod("chmod.dat", S_IREAD))
         perror("Chmod failed");
      if (-1 == access("chmod.dat", 02))
         printf("File chmod.dat is now readonly.\n\n");
      printf("Run this program again to erase chmod.dat.\n\n");
   }
   else {
      printf("\nFile chmod.dat exist.\n");
      printf("chmod chmod.dat to become writable.\n");
      if (-1 == chmod("chmod.dat", S_IWRITE))
         perror("Chmod failed");
      system("erase  chmod.dat");
      printf("File chmod.dat removed.\n\n");
   }
   return 0;

   /***************************************************************************
      If chmod.dat does not exist, the output should be :

       Creating chmod.dat.
       chmod chmod.dat to be readonly.
       File chmod.dat is now readonly.

       Run this program again to erase chmod.dat.
   ****************************************************************************/
}
```

Related Information

- access
- _sopen
- umask

_chsize - Alter Length of File

_chsize - Alter Length of File

Syntax

```
#include <io.h>
int _chsize(int handle, long size);
```

Description

_chsize lengthens or cuts off the file associated with *handle* to the length specified by *size* . You must open the file in a mode that permits writing. _chsize adds null characters (`\0`) when it lengthens the file. When _chsize cuts off the file, it erases all data from the end of the shortened file to the end of the original file.

Returns

_chsize returns the value `0` if it successfully changes the file size. A return value of `-1` shows an error; _chsize sets `errno` to one of the following values:

Value	**Meaning**
EACCESS	The specified file is locked against access.
EBADF	The file handle is not valid, or the file is not open for writing.
ENOSPC	There is no space left on the device.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example opens a file named `sample.dat` and returns the current length of that file. It then alters the size of `sample.dat` and returns the new length of that file.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
   long length;
   int fh;

   printf("\nCreating sample.dat.\n");
   system("echo Sample Program > sample.dat");
   if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
      printf("Unable to open sample.dat.\n");
      return EXIT_FAILURE;
   }
   if (-1 == (length = filelength(fh))) {
      printf("Unable to determine length of sample.dat.\n");
      return EXIT_FAILURE;
   }
   printf("Current length of sample.dat is %d.\n", length);
   printf("Changing the length of sample.dat to 20.\n");
   if (-1 == (_chsize(fh, 20))) {
      perror("chsize failed");
      return EXIT_FAILURE;
   }
   if (-1 == (length = _filelength(fh))) {
      printf("Unable to determine length of sample.dat.\n");
      return EXIT_FAILURE;
   }
   printf("New length of sample.dat is %d.\n", length);
   close(fh);
   return 0;

   /****************************************************************************
      The output should be similar to :
       Creating sample.dat.
       Current length of sample.dat is 17.
       Changing the length of sample.dat to 20.
       New length of sample.dat is 20.
   ****************************************************************************/
}
```

Related Information

- _filelength
- lseek

----

clearerr - Reset Error Indicators

clearerr - Reset Error Indicators

Syntax

```
#include <stdio.h>
void clearerr (FILE *stream);
```

Description

The `clearerr` function resets the error indicator and end-of-file indicator for the specified *stream* . Once set, the indicators for a specified stream remain set until your program calls `clearerr` or `rewind`. fseek also clears the end-of-file indicator.

Returns

There is no return value.

Example Code

This example reads a data stream and then checks that a read error has not occurred.

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;

int main(void)
{
   if (NULL != (stream = fopen("file.dat", "r"))) {
      if (EOF == (c = getc(stream))) {
         if (feof(stream)) {
            perror("Read error");
            clearerr(stream);
         }
      }
   }
   return 0;

   /****************************************************************************
      If file.dat is an empty file, the output should be:

      Read error: Attempted to read past end-of-file.

   ****************************************************************************/
}
```

Related Information

- feof
- ferror
- perror
- rewind
- strerror
- _strerror

----

clock - Determine Processor Time

clock - Determine Processor Time

Syntax

```
#include <time.h>
clock_t clock(void);
```

Description

`clock` returns an approximation of the processor time used by the program since the beginning of an implementation-defined time-period that is related to the program invocation. To obtain the time in seconds, divide the value returned by `clock` by the value of the macro `CLOCKS_PER_SEC`.

Returns

If the value of the processor time is not available or cannot be represented, `clock` returns the value `(clock_t)-1`.

To measure the time spent in a program, call `clock` at the start of the program, and subtract its return value from the value returned by subsequent calls to `clock`.

Example Code

This example prints the time elapsed since the program was invoked.

```
#include <time.h>
#include <stdio.h>

double time1,time2,timedif;               /* use doubles to show small values */
int i;

int main(void)
{
   time1 = (double)clock();               /* get initial time in seconds      */
   time1 = time1/CLOCKS_PER_SEC;

   /* use some CPU time                                                       */

   for (i = 0; i < 5000000; i++) {
      int j;

      j = i;
   }
   time2 = (double)clock();                /* call clock a second time        */
   time2 = time2/CLOCKS_PER_SEC;
   timedif = time2-time1;
   printf("The elapsed time is %f seconds\n", timedif);
   return 0;

   /****************************************************************************
```

```
      The output should be similar to:

      The elapsed time is 0.969000 seconds
   ****************************************************************************/
}
```

Related Information

- difftime
- time

----

close - Close File Associated with Handle

close - Close File Associated with Handle

Syntax

```
#include <io.h>
int close(int handle);
```

Description

close closes the file associated with the handle. Use close if you opened the handle with open. If you opened the handle with fopen, use fclose to close it.

Returns

close returns `0` if it successfully closes the file. A return value of `-1` shows an error, and close sets `errno` to EBADF, showing an incorrect file handle argument.

Example Code

This example opens the file `edclose.dat` and then closes it using the close function.

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>

int main(void)
{
   int fh;

   printf("\nCreating edclose.dat.\n");
   if (-1 == (fh = open("edclose.dat", O_RDWR|O_CREAT|O_TRUNC, S_IREAD|S_IWRITE)
      )) {
      perror("Unable to open edclose.dat");
      return EXIT_FAILURE;
   }
   printf("File was successfully opened.\n");
   if (-1 == close(fh)) {
      perror("Unable to close edclose.dat");
      return EXIT_FAILURE;
   }
   printf("File was successfully closed.\n");
   return 0;

   /****************************************************************************
      The output should be:

      Creating edclose.dat.
      File was successfully opened.
      File was successfully closed.
   ****************************************************************************/
}
```

Related Information

- fclose
- creat
- open
- _sopen

---

cos - Calculate Cosine

cos - Calculate Cosine

Syntax

```
#include <math.h>
double cos(double x);
```

Description

`cos` calculates the cosine of *x*. The value *x* is expressed in radians. If *x* is too large, a partial loss of significance in the result may occur.

Returns

`cos` returns the cosine of *x*.

Example Code

This example calculates *y* to be the cosine of *x*.

```
#include <math.h>

int main(void)
{
   double x,y;

   x = 7.2;
   y = cos(x);
   printf("cos( %lf ) = %lf\n", x, y);
   return 0;

   /****************************************************************************
      The output should be:

      cos( 7.200000 ) = 0.608351
   ****************************************************************************/
}
```

Related Information

- acos
- cosh
- sin
- sinh
- tan
- tanh

----

cosh - Calculate Hyperbolic Cosine

cosh - Calculate Hyperbolic Cosine

Syntax

```
#include <math.h>
double cosh(double x);
```

Description

`cosh` calculates the hyperbolic cosine of *x*. The value *x* is expressed in radians.

Returns

`cosh` returns the hyperbolic cosine of *x*. If the result is too large, `cosh` returns the value `HUGE_VAL` and sets `errno` to `ERANGE`.

Example Code

This example calculates *y* to be the hyperbolic cosine of *x*.

```
#include <math.h>

int main(void)
{
   double x,y;

   x = 7.2;
   y = cosh(x);
   printf("cosh( %lf ) = %lf\n", x, y);
   return 0;

   /****************************************************************************
     The output should be:

     cosh( 7.200000 ) = 669.715755
   ****************************************************************************/
}
```

Related Information

- acos
- cos
- sin
- sinh
- tan
- tanh

_cprintf - Print Characters to Screen

_cprintf - Print Characters to Screen

Syntax

```
#include <conio.h>
int _cprintf(char *format-string, argument-list);
```

Description

_cprintf formats and sends a series of characters and values directly to the screen, using the _putch function to send each character.

The *format-string* has the same form and function as the *format-string* parameter for printf. Format specifications in the *format-string* determine the output format for any *argument-list* that follows. See printf for a description of the *format-string* .

Note: Unlike the fprintf and printf functions, _cprintf does not translate line feed characters into output of a carriage return followed by a line feed.

Returns

_cprintf returns the number of characters printed.

Example Code

The following program uses _cprintf to write strings to the screen.

```
#include <conio.h>

int main(void)
{
   char buffer[24];

   _cprintf("\nPlease enter a filename:\n");
   _cscanf("%23s", buffer);
   _cprintf("\nThe file name you entered was %23s.", buffer);
   return 0;

   /****************************************************************************
      The output should be similar to :

       Please enter a filename:
                              file.dat
       The filename you entered was                file.dat.
   ****************************************************************************/
}
```

Related Information

- _cscanf
- fprintf
- printf
- _putch
- sprintf

_cputs - Write String to Screen

_cputs - Write String to Screen

Syntax

```
#include <conio.h>
int _cputs(char *str);
```

Description

_cputs writes the string that *str* points to directly to the screen. The string *str* must end with a null character (\0). The _cputs function does not automatically add a carriage return followed by a line feed to the string.

Returns

If successful, _cputs returns 0. Otherwise, it returns a nonzero value.

Example Code

This example outputs a prompt to the screen.

```
#include <conio.h>

int main(void)
{
   char *buffer = "Insert data disk in drive a: \r\n";

   _cputs(buffer);
   return 0;

   /****************************************************************************
      The output should be:

      Insert data disk in drive a:
   ****************************************************************************/
}
```

Related Information

- _cgets
- fputs
- _putch
- puts

----

creat - Create New File

creat - Create New File

Syntax

```
#include <io.h>
#include <sys\stat.h>
int creat(char *pathname, int pmode);
```

Description

creat either creates a new file or opens and truncates an existing file. If the file specified by *pathname* does not exist, creat creates a new file with the given permission setting and opens it for writing in text mode. If the file already exists, and the read-only attribute and sharing permissions allow writing, creat truncates the file to length 0. This action destroys the previous contents of the file and opens it for writing in text mode. creat always opens a file in text mode for reading and writing.

The permission setting *pmode* applies to newly created files only. The new file receives the specified permission setting after you close it for the first time. The *pmode* integer expression contains one or both of the constants S_IWRITE and S_IREAD, defined in <sys\stat.h>. The values of the *pmode* argument and their meanings are: compact break=fit.

Value	Meaning
S_IREAD	Reading permitted
S_IWRITE	Writing permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If you do not give permission to write to the file, it is a read-only file. On the OS/2 operating system, you cannot give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE have the same results. The creat function applies the current file permission mask to *pmode* before setting the permissions. (See umask for more information about file permission masks.)

When writing new code, you should use open rather than creat.

Specifying a *pmode* of S_IREAD is similar to making a file read-only with the ATTRIB system command.

Returns

creat returns a handle for the created file if the call is successful. A return value of `-1` shows an error, and creat sets `errno` to one of the following values:

Value	Meaning
EACCESS	File sharing violated.
EINVAL	The mode specified was not valid.
EMFILE	No more file handles are available.
ENOENT	The path name was not found, or the file name was incorrect.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example creates the file `sample.dat` so it can be read from and written to.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int fh;

   fh = creat("sample.dat", S_IREAD|S_IWRITE);
   if (-1 == fh) {
      perror("Error in creating sample.dat");
      return EXIT_FAILURE;
   }
   else
      printf("Successfully created sample.dat.\n");
   close(fh);
   return 0;

   /****************************************************************************
      The output should:

       Successfully created sample.dat.
   ****************************************************************************/
}
```

Related Information

- chmod
- close
- open
- fdopen
- _sopen
- umask

_CRT_init - Initialize DLL Run-Time Environment

_CRT_init - Initialize DLL Run-Time Environment

Syntax

```
int _CRT_init(void);
/* no header file - defined in the run time startup code */
```

Description

_CRT_init initializes *The Developer's Toolkit* run-time environment for a DLL.

By default, all DLLs call *The Developer's Toolkit* _DLL_InitTerm function, which in turn calls _CRT_init for you. However, if you are writing your own _DLL_InitTerm function (for example, to perform actions other than run time initialization and termination), you must call _CRT_init from your version of _DLL_InitTerm before you can call any other run-time functions.

If your DLL contains C++ code, you must also call `__ctordtorInit` after _CRT_init to ensure that static constructors and destructors are initialized properly. __ctordtorInit is defined in the run-time startup code as:

```
   void __ctordtorInit(void);
```

Note: If you are providing your own version of the _matherr function to be used in your DLL, you must also call the `_exception_dllinit` function after the run-time environment is initialized. Calling this function ensures that the proper _matherr function will be called during exception handling. `_exception_dllinit` is defined in the run-time startup code as:

```
   void _exception_dllinit( int (*)(struct exception *) );
```

The parameter required is the address of your _matherr function.

Returns

If the run-time environment is successfully initialized, _CRT_init returns `0`. A return code of `-1` indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of stderr.

Example Code

The following example shows the _DLL_InitTerm function from *The Developer's Toolkit* sample program for building DLLs, which calls _CRT_init to initialize the library environment.

```
#define  INCL_DOSMODULEMGR
#define  INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* _CRT_init is the C run-time environment initialization function.         */
/* It will return 0 to indicate success and -1 to indicate failure.         */

int _CRT_init(void);
#ifdef   STATIC_LINK

/* _CRT_term is the C run-time environment termination function.            */
/* It only needs to be called when the C run-time functions are statically  */
/* linked.                                                                  */
```

```
void _CRT_term(void);
#else

/* A clean up routine registered with DosExitList must be used if run-time  */
/* calls are required and the run time is dynamically linked.  This will    */
/* guarantee that this clean up routine is run before the library DLL is    */
/* terminated.                                                              */

static void cleanup(ULONG ulReason);
#endif
size_t nSize;
int *pArray;

/* _DLL_InitTerm is the function that gets called by the operating system   */
/* loader when it loads and frees this DLL for each process that accesses   */
/* this DLL.  However, it only gets called the first time the DLL is loaded */
/* and the last time it is freed for a particular process.  The system      */
/* linkage convention MUST be used because the operating system loader is   */
/* calling this function.                                                   */

unsigned long _DLL_InitTerm(unsigned long hModule, unsigned long
                                    ulFlag)
{
   size_t i;
   APIRET rc;
   char namebuf[CCHMAXPATH];

   /* If ulFlag is zero then the DLL is being loaded so initialization should*/
   /* be performed.  If ulFlag is 1 then the DLL is being freed so          */
   /* termination should be performed.                                      */

   switch (ulFlag) {
      case 0 :

         /*******************************************************************/
         /* The C run-time environment initialization function must be      */
         /* called before any calls to C run-time functions that are not    */
         /* inlined.                                                        */
         /*******************************************************************/

         if (_CRT_init() == -1)
            return 0UL;



#ifndef  STATIC_LINK

       /*********************************************************************/
       /* A DosExitList routine must be used to clean up if run-time calls */
       /* are required and the run time is dynamically linked.             */
       /*********************************************************************/

            if (rc = DosExitList(0x0000FF00|EXLST_ADD, cleanup))
            printf("DosExitList returned %lu\n", rc);
#endif
         if (rc = DosQueryModuleName(hModule, CCHMAXPATH, namebuf))
            printf("DosQueryModuleName returned %lu\n", rc);
         else
            printf("The name of this DLL is %s\n", namebuf);
         srand(17);
         nSize = (rand()%128)+32;
         printf("The array size for this process is %u\n", nSize);
         if ((pArray = malloc(nSize *sizeof(int))) == NULL) {
            printf("Could not allocate space for unsorted array.\n");
            return 0UL;
         }
         for (i = 0; i < nSize; ++i)
            pArray[i] = rand();
         break;
      case 1 :
#ifdef   STATIC_LINK
         printf("The array will now be freed.\n");
         free(pArray);
         _CRT_term();
#endif
         break;
      default  :
         printf("ulFlag = %lu\n", ulFlag);
         return 0UL;
   }

   /* A non-zero value must be returned to indicate success.                */
   return 1UL;
}
#ifndef  STATIC_LINK
static void cleanup(ULONG ulReason)
{
   if (!ulReason) {
      printf("The array will now be freed.\n");
      free(pArray);
   }
   DosExitList(EXLST_EXIT, cleanup);
   return ;
}
#endif
```

Related Information

- "Building Dynamic Link Libraries" in the *VisualAge C++ Programming Guide*
- _CRT_term
- _DLL_InitTerm
- _rmem_init
- _rmem_term

_CRT_term - Terminate DLL Run-Time Environment

_CRT_term - Terminate DLL Run-Time Environment

Syntax

```
void _CRT_term(void);
/* no header file - defined in run-time startup code */
```

Description

_CRT_term terminates *The Developer's Toolkit* run-time environment. It is only needed for DLLs where the C run-time functions are statically linked.

By default, all DLLs call *The Developer's Toolkit* _DLL_InitTerm function, which in turn calls _CRT_term for you. However, if you are writing your own _DLL_InitTerm function (for example, to perform actions other than run time initialization and termination), and your DLL statically links to the C run-time libraries, you need to call _CRT_term from your _DLL_InitTerm function.

If your DLL contains C++ code, you must also call `__ctordtorTerm` **before** you call _CRT_term to ensure that static constructors and destructors are terminated correctly. __ctordtorTerm is defined in the run-time startup code as:

```
   void __ctordtorTerm(void);
```

Once you have called _CRT_term, you cannot call any other library functions.

If your DLL is dynamically linked, you cannot call library functions in the termination section of your _DLL_InitTerm function. If your termination routine requires calling library functions, you must register the termination routine with DosExitList. Note that all DosExitList routines are called before DLL termination routines.

Returns

There is no return value for _CRT_term.

Example Code

The following example shows the _DLL_InitTerm function from *The Developer's Toolkit* sample program for building DLLs, which calls _CRT_term to terminate the library environment.

```
#define  INCL_DOSMODULEMGR
#define  INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* _CRT_init is the C run-time environment initialization function.         */
/* It will return 0 to indicate success and -1 to indicate failure.         */

int _CRT_init(void);
#ifdef   STATIC_LINK

/* _CRT_term is the C run-time environment termination function.            */
/* It only needs to be called when the C run-time functions are statically  */
/* linked.                                                                  */

void _CRT_term(void);
#else

/* A clean up routine registered with DosExitList must be used if run-time  */
/* calls are required and the run time is dynamically linked.  This will    */
/* guarantee that this clean up routine is run before the library DLL is    */
/* terminated.                                                              */

static void cleanup(ULONG ulReason);
#endif
size_t nSize;
int *pArray;

/* _DLL_InitTerm is the function that gets called by the operating system   */
/* loader when it loads and frees this DLL for each process that accesses   */
/* this DLL.  However, it only gets called the first time the DLL is loaded */
/* and the last time it is freed for a particular process.  The system      */
/* linkage convention MUST be used because the operating system loader is   */
/* calling this function.                                                   */

unsigned long _DLL_InitTerm(unsigned long hModule, unsigned long
                                    ulFlag)
{
   size_t i;
   APIRET rc;
   char namebuf[CCHMAXPATH];

   /* If ulFlag is zero then the DLL is being loaded so initialization should*/
   /* be performed.  If ulFlag is 1 then the DLL is being freed so          */
   /* termination should be performed.                                      */

   switch (ulFlag) {
      case 0 :

         /*******************************************************************/
         /* The C run-time environment initialization function must be      */
         /* called before any calls to C run-time functions that are not    */
         /* inlined.                                                        */
         /*******************************************************************/

         if (_CRT_init() == -1)
            return 0UL;



#ifndef  STATIC_LINK
         /********************************************************************/
         /* A DosExitList routine must be used to clean up if run-time calls */
         /* are required and the run time is dynamically linked.             */
         /********************************************************************/

            if (rc = DosExitList(0x0000FF00|EXLST_ADD, cleanup))
            printf("DosExitList returned %lu\n", rc);
#endif
         if (rc = DosQueryModuleName(hModule, CCHMAXPATH, namebuf))
            printf("DosQueryModuleName returned %lu\n", rc);
         else
            printf("The name of this DLL is %s\n", namebuf);
         srand(17);
         nSize = (rand()%128)+32;
         printf("The array size for this process is %u\n", nSize);
         if ((pArray = malloc(nSize *sizeof(int))) == NULL) {
            printf("Could not allocate space for unsorted array.\n");
            return 0UL;
         }
         for (i = 0; i < nSize; ++i)
            pArray[i] = rand();
         break;
      case 1 :
#ifdef   STATIC_LINK
         printf("The array will now be freed.\n");
         free(pArray);
         _CRT_term();
#endif
         break;
      default  :
         printf("ulFlag = %lu\n", ulFlag);
         return 0UL;
   }

   /* A non-zero value must be returned to indicate success.                 */

   return 1UL;
}
#ifndef  STATIC_LINK
static void cleanup(ULONG ulReason)
{
   if (!ulReason) {
      printf("The array will now be freed.\n");
      free(pArray);
   }
   DosExitList(EXLST_EXIT, cleanup);
   return ;
}
#endif
```

Related Information

- "Building Dynamic Link Libraries" in the *VisualAge C++ Programming Guide*
- _CRT_init
- _DLL_InitTerm
- _rmem_init
- _rmem_term

_cscanf - Read Data from Keyboard

_cscanf - Read Data from Keyboard

Syntax

```
#include <conio.h>
int _cscanf(char *format-string, argument-list);
```

Description

_cscanf reads data directly from the keyboard to the locations given by *argument-list* , if any are specified. The _cscanf function uses the _getche function to read characters. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string* .

The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the scanf function. See scanf for a description of the *format-string* .

Note: Although _cscanf normally echoes the input character, it does not do so if the last action was a call to _ungetch.

Returns

_cscanf returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at the end of the file. A return value of `0` means that no fields were assigned.

Example Code

This example uses `_cscanf` to read strings from the screen.

```
#include <conio.h>

int main(void)
{
   char buffer[24];

   _cprintf("\nPlease enter a filename:\n");
   _cscanf("%23s", buffer);
   _cprintf("\nThe file name you entered was %23s.", buffer);
   return 0;

   /****************************************************************************
      The output should be similar to :

      Please enter a filename:
                              file.dat
      The filename you entered was                file.dat.
   ****************************************************************************/
}
```

Related Information

- fscanf
- _getch - _getche
- scanf
- sscanf
- _ungetch

----

ctime - Convert Time to Character String

ctime - Convert Time to Character String

Syntax

```
#include <time.h>
char *ctime(const time_t *time);
```

Description

`ctime` converts the time value pointed to by *time* to local time in the form of a character string. A time value is usually obtained by a call to the `time` function.

The string result produced by `ctime` contains exactly 26 characters and has the format:

```
   "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

For example:

```
   Mon Jul 16 02:03:55 1987\n\0
```

`ctime` uses a 24-hour clock format. The days are abbreviated to: `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, and `Sat`. The months are abbreviated to: `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, and `Dec`. All fields have a constant width. Dates with only one digit are preceded with a blank space. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Returns

`ctime` returns a pointer to the character string result. There is no error return value. A call to `ctime` is equivalent to:

```
   asctime(localtime(&anytime))
```

Note: The `asctime`, `ctime`, and other time functions may use a common, statically allocated buffer for holding the return string. Each call to one of these functions may destroy the result of the previous call.

Example Code

This example polls the system clock using ctime. It then prints a message giving the current date and time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;

   time(&ltime);
   printf("The time is %s", ctime(&ltime));
   return 0;

   /****************************************************************************
      The output should be similar to :

      The time is Thu Dec 15 18:10:23 1994
   ****************************************************************************/
}
```

Related Information

- asctime
- gmtime
- localtime
- mktime
- strftime
- time

_cwait - Wait for Child Process

_cwait - Wait for Child Process

Syntax

```
#include <process.h>
int _cwait(int *stat_loc, int process_id, int action_code);
```

Description

_cwait delays the completion of a parent process until the child process specified by *process_id* ends.

The *process_id* is the value returned by the _spawn function that started the child process. If the specified child process ends before _cwait is called, _cwait returns to the calling process immediately with a value of -1. If the value of *process_id* is `0`, the parent process waits until all of its child processes end.

If the variable pointed to by *stat_loc* is `NULL`, _cwait does not use it. If it is not `NULL`, _cwait places information about the return status and the return code of the child process in the location pointed to by *stat_loc* .

If the child process ended normally with a call to the OS/2 DosExit function, the lowest-order byte of the variable pointed to by *stat_loc* is `0`. The next highest-order byte contains the lowest-order byte of the argument passed to DosExit by the child process. The value of this byte depends on how the child process caused the system to call DosExit. If the child called exit, _exit, or `return` from `main`, or used a DosExit coded into the program, the byte contains the lowest-order byte of the argument the child passed to exit, _exit, or `return`. The value of the byte is undefined if the child caused a DosExit call simply by reaching the end of `main`.

If the child process ended abnormally (without a call to DosExit), the lowest-order byte of the variable pointed to by *stat_loc* contains the return code from the OS/2 DosWaitChild function, and the next higher-order byte is `0`. See the OS/2 online reference for details about the DosWaitChild return codes.

The *action_code* specifies when the parent process is to start running again. Values for *action_code* include:

Action Code	**Meaning**
WAIT_CHILD	The parent process waits until the specified child process ends.
WAIT_GRANDCHILD	The parent process waits until the child process and all of the child processes of that process end.

The action code values are defined in `<process.h>`.

An alternative to this function is the DosWaitChild call.

Returns

At the normal end of the child process, _cwait returns the process identifier of the child to the parent process. If a child process ends abnormally, _cwait returns -1 to the parent process and sets `errno` to EINTR. In the case of an error, _cwait returns immediately with a value of -1 and sets `errno` to one of the following values:

Value	**Meaning**
EINVAL	Incorrect action code.
ECHILD	No child process exists, or the process identifier is incorrect.

Example Code

This example creates a new process called `child.exe`. The parent calls _cwait and waits for the child to end. The parent then displays the child's return information in hexadecimal.

```
#include <stdio.h>
#include <process.h>
#include <errno.h>

int stat_child;

int main(void)
{
   int i,result;

   /* spawn a child and 'cwait' for it to finish                              */

   if ((result = _spawnl(P_NOWAIT, "child", "child", "1", NULL)) != -1) {
      if ((i = _cwait(&stat_child, result, WAIT_CHILD)) != result)
         printf("Error ...expected pid from child");
      else {
         if (0 == errno) {
            printf("Child process ended successfully and ...\n");
            printf("program returned to the Parent process.\n");
         }
         else
            printf("Child process had an error\n");
      }
   }
   else
      printf("Error ...could not spawn a child process\n");
   return 0;
   /****************************************************************************

      If the source code for child.exe is:

      #include <stdio.h>

      int main(void) {

         puts("This line was written by child.exe");
         return 0;
      }

      The output should be similar to :

      This line was written by child.exe
      Child process ended successfully and ...
      program returned to the Parent process.
   ****************************************************************************/
}
```

Related Information

- exit
- _exit
- _spawnl - _spawnvpe
- wait
- return

----

difftime - Compute Time Difference

difftime - Compute Time Difference

Syntax

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Description

`difftime` computes the difference in seconds between *time2* and *time1* .

Returns

The `difftime` function returns the elapsed time in seconds from *time1* to *time2* as a double precision number. Type `time_t` is defined in `<time.h>`.

Example Code

This example shows a timing application using `difftime`. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10000.

```
#include <time.h>
#include <stdio.h>

#define  RUNS          1000
#define  SIZE          10000

int mark[SIZE];

int main(void)
{
   time_t start,finish;
   int i,loop,n,num;

   time(&start);


   /*  This loop finds the prime numbers between 2 and SIZE                   */

   for (loop = 0; loop < RUNS; ++loop) {
      for (n = 0; n < SIZE; ++n)
         mark[n] = 0;

      /*  This loops marks all the composite numbers with -1                  */

      for (num = 0, n = 2; n < SIZE; ++n)
         if (!mark[n]) {
            for (i = 2*n; i < SIZE; i += n)
               mark[i] = -1;
            ++num;
         }
   }
   time(&finish);
   printf("Program takes an average of %f seconds to find %d primes.\n",
      difftime(finish, start)/RUNS, num);
   return 0;

   /****************************************************************************
      The output should be similar to :

      Program takes an average of 0.106000 seconds to find 1229 primes.
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- localtime
- mktime
- strftime
- time

----

div - Calculate Quotient and Remainder

div - Calculate Quotient and Remainder

Syntax

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

Description

`div` calculates the quotient and remainder of the division of *numerator* by *denominator*.

Returns

`div` returns a structure of type `div_t`, containing both the quotient `int quot` and the remainder `int rem`. If the return value cannot be represented, its value is undefined. If *denominator* is `0`, an exception will be raised.

Example Code

This example uses `div` to calculate the quotients and remainders for a set of two dividends and two divisors.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int num[2] =  { 45,-45 };
   int den[2] =  { 7,-7 };
   div_t ans;              /* div_t is a struct type containing two ints:
                              'quot' stores quotient; 'rem' stores remainder */
   short i,j;

   printf("Results of division:\n");
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++) {
         ans = div(num[i], den[j]);
         printf("Dividend: %6ld  Divisor: %6ld", num[i], den[j]);
         printf("  Quotient: %6ld  Remainder: %6ld\n", ans.quot, ans.rem);
      }
   return 0;

   /****************************************************************************
      The output should be:

      Results of division:
      Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
      Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
      Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
      Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3
   ****************************************************************************/
}
```

Related Information

- ldiv

_DLL_InitTerm - Initialize and Terminate DLL Environment

_DLL_InitTerm - Initialize and Terminate DLL Environment

Syntax

```
unsigned long _DLL_InitTerm(unsigned long modhandle,
                                    unsigned long flag);
/* no header file - defined in run-time startup code */
```

Description

_DLL_InitTerm is the initialization and termination entry point for a DLL. When each new process gains access to the DLL, _DLL_InitTerm initializes the necessary environment for the DLL, including storage, semaphores, and variables. When each process frees its access to the DLL, _DLL_InitTerm terminates the DLL environment created for that process.

The default _DLL_InitTerm function performs the actions required to initialize and terminate the run-time environment, or for subsystem DLLs, to initialize and terminate memory functions. It is called automatically when you link to the DLL. If your DLL requires initialization or termination actions in addition to the actions performed in the default function, you will need to create your own _DLL_InitTerm function.

If the value of the *flag* parameter is 0, the DLL environment is initialized. If the value of the *flag* parameter is 1, the DLL environment is ended.

The *modhandle* parameter is the module handle assigned by the operating system for this DLL. You can use the module handle as a parameter to various OS/2 function calls. For example, you can call DosQueryModuleName with the module handle to return the fully-qualified path name of the DLL, which tells you where the DLL was loaded from.

For more information on creating your own _DLL_InitTerm function, see the chapter "Building Dynamic Link Libraries in the *VisualAge C++ Programming Guide* .

Note: A _DLL_InitTerm function for a subsystem DLL has the same prototype, but the content of the function is different because there is no run-time environment to initialize or terminate. For more information on building subsystem DLLs, see the section "Building a Subsystem DLL" in the *VisualAge C++ Programming Guide* .

Returns

The return code from _DLL_InitTerm tells the loader if the initialization or termination was performed successfully. If the call was successful, _DLL_InitTerm returns a nonzero value. A return code of `0` indicates that the function failed. If a failure is indicated, the loader will not load the program that is accessing the DLL.

Example Code

The following example shows the _DLL_InitTerm function sample program for building DLLs.

```
#define  INCL_DOSMODULEMGR
#define  INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* _CRT_init is the C run-time environment initialization function.         */
/* It will return 0 to indicate success and -1 to indicate failure.         */

int _CRT_init(void);
#ifdef   STATIC_LINK

/* _CRT_term is the C run-time environment termination function.            */
/* It only needs to be called when the C run-time functions are statically  */
/* linked.                                                                  */

void _CRT_term(void);
#else

/* A clean up routine registered with DosExitList must be used if run-time  */
/* calls are required and the run time is dynamically linked.  This will    */
/* guarantee that this clean up routine is run before the library DLL is    */
/* terminated.                                                              */

static void cleanup(ULONG ulReason);
#endif
size_t nSize;
int *pArray;

/* _DLL_InitTerm is the function that gets called by the operating system   */
/* loader when it loads and frees this DLL for each process that accesses   */
/* this DLL.  However, it only gets called the first time the DLL is loaded */
/* and the last time it is freed for a particular process.  The system      */
/* linkage convention MUST be used because the operating system loader is   */
/* calling this function.                                                   */

unsigned long _DLL_InitTerm(unsigned long hModule, unsigned long
                                    ulFlag)
{
   size_t i;
   APIRET rc;
   char namebuf[CCHMAXPATH];
   /* If ulFlag is zero then the DLL is being loaded so initialization should*/
   /* be performed.  If ulFlag is 1 then the DLL is being freed so          */
   /* termination should be performed.                                      */



   switch (ulFlag) {
      case 0 :

         /*******************************************************************/
         /* The C run-time environment initialization function must be      */
         /* called before any calls to C run-time functions that are not    */
         /* inlined.                                                        */
         /*******************************************************************/

         if (_CRT_init() == -1)
            return 0UL;
#ifndef  STATIC_LINK

         /********************************************************************/
         /* A DosExitList routine must be used to clean up if run-time calls */
         /* are required and the run time is dynamically linked.             */
         /********************************************************************/

            if (rc = DosExitList(0x0000FF00|EXLST_ADD, cleanup))
            printf("DosExitList returned %lu\n", rc);
#endif
         if (rc = DosQueryModuleName(hModule, CCHMAXPATH, namebuf))
            printf("DosQueryModuleName returned %lu\n", rc);
         else
            printf("The name of this DLL is %s\n", namebuf);
         srand(17);
         nSize = (rand()%128)+32;
         printf("The array size for this process is %u\n", nSize);
         if ((pArray = malloc(nSize *sizeof(int))) == NULL) {
            printf("Could not allocate space for unsorted array.\n");
            return 0UL;
         }
         for (i = 0; i < nSize; ++i)
            pArray[i] = rand();
         break;
      case 1 :
#ifdef   STATIC_LINK
         printf("The array will now be freed.\n");
         free(pArray);
         _CRT_term();
#endif
         break;
      default  :
         printf("ulFlag = %lu\n", ulFlag);
         return 0UL;
   }
   /* A non-zero value must be returned to indicate success.                */

   return 1UL;
}
#ifndef  STATIC_LINK
static void cleanup(ULONG ulReason)
{
   if (!ulReason) {
      printf("The array will now be freed.\n");
      free(pArray);
   }
   DosExitList(EXLST_EXIT, cleanup);
   return ;
}
#endif
```

The following example shows the _DLL_InitTerm function sample for building subsystem DLLs.

```
/* _DLL_InitTerm() - called by the loader for DLL
initialization/termination  */
/* This function must return a non-zero value if successful and a zero value  */
/* if unsuccessful.                                                           */

unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag )
   {
   APIRET rc;

   /* If ulFlag is zero then initialization is required:                      */
   /*    If the shared memory pointer is NULL then the DLL is being loaded    */
   /*    for the first time so acquire the named shared storage for the       */
   /*    process control structures.  A linked list of process control        */
   /*    structures will be maintained.  Each time a new process loads this   */
   /*    DLL, a new process control structure is created and it is inserted   */
   /*    at the end of the list by calling DLLREGISTER.                       */
   /*                                                                         */
   /* If ulFlag is 1 then termination is required:                            */
   /*    Call DLLDEREGISTER which will remove the process control structure   */
   /*    and free the shared memory block from its virtual address space.     */

   switch( ulFlag )
      {
      case 0:
         if ( !ulProcessCount )
            {
```

```
         /* Create the shared mutex semaphore.                             */

         if ( ( rc = DosCreateMutexSem( SHARED_SEMAPHORE_NAME,
                                        &hmtxSharedSem,
                                        0,
                                        FALSE ) ) != NO_ERROR )
            {
            printf( "DosCreateMutexSem rc = %lu\n", rc );
            return 0;
            }
         }
      /* Register the current process.                                     */

      if ( DLLREGISTER( ) )
         return 0;
      break;

   case 1:
      /* De-register the current process.                                  */

      if ( DLLDEREGISTER( ) )
         return 0;
      break;

   default:
      return 0;
   }

/* Indicate success.  Non-zero means success!!!             */

return 1;
}
```

Related Information

- "Building a Dynamic Link Library" in the *VisualAge C++ Programming Guide*
- "Building a Subsystem DLL" in the *VisualAge C++ Programming Guide*
- _CRT_init
- _CRT_term
- _rmem_init
- _rmem_term

----

dup - Associate Second Handle with Open File

dup - Associate Second Handle with Open File

Syntax

```
#include <io.h>
int dup(int handle);
```

Description

dup associates a second file handle with a currently open file. You can carry out operations on the file using either file handle because all handles associated with a given file use the same file pointer. Creation of a new handle does not affect the type of access allowed for the file.

For example, given:

```
   handle2 = dup(handle1)
```

handle2 will have the same file access mode (text or binary) as *handle1* . In addition, if *handle1* was originally opened with the O_APPEND flag (described in open), *handle2* will also have that attribute.

Warning: Both handles share a single file pointer. If you reposition a file using *handle1* , the position in the file returned by *handle2* will also change.

If you duplicate a file handle for an open stream, the resulting file handle has the same restrictions as the original file handle.

Returns

dup returns the next available file handle for the given file. It returns `-1` if an error occurs and sets `errno` to one of the following values:

Value	**Meaning**
EBADF	The file handle is not valid.
EMFILE	No more file handles are available.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example makes a second file handle, `fh3`, refer to the same file as the file handle `fh1` using dup. The file handle `fh2` is then associated with the file `edopen.da2`, and finally `fh2` is forced to associate with `edopen.da1` `dup2`.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>

int main(void)
{
   int fh1,fh2,fh3;

   if (-1 == (fh1 = open("edopen.da1", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
      )) {
      perror("Unable to open edopen.da1");
      return EXIT_FAILURE;
   }
   if (-1 == (fh3 = dup(fh1))) {     /* fh3 refers to the sample file as fh1  */
      perror("Unable to dup");
      close(fh1);
      return EXIT_FAILURE;
   }
   else
      printf("Successfully performed dup handle.\n");
   if (-1 == (fh2 = open("edopen.da2", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
      )) {
      perror("Unable to open edopen.da2");
      close(fh1);
      close(fh3);
      return EXIT_FAILURE;
   }
   if (-1 == dup2(fh1, fh2)) {  /* Force fh2 to the refer to the same file     */
                                /* as fh1.                                    */
      perror("Unable to dup2");
   }
   else
      printf("Successfully performed dup2 handle.\n");
   close(fh1);
   close(fh2);
   close(fh3);
   return 0;

   /****************************************************************************
      The output should be:

       Successfully performed dup handle.
       Successfully performed dup2 handle.
   ****************************************************************************/
}
```

Related Information

- close

- creat
- dup2
- open
- _sopen

----

dup2 - Associate Second Handle with Open File

dup2 - Associate Second Handle with Open File

Syntax

```
#include <io.h>
int dup2(int handle1, int handle2);
```

Description

dup2 makes *handle2* refer to the currently open file associated with *handle1* . You can carry out operations on the file using either file handle because all handles associated with a given file use the same file pointer.

handle2 will point to the same file as *handle1* , but will retain the file access mode (text or binary) that it had before duplication. In addition, if *handle2* was originally opened with the O_APPEND flag, it will also retain that attribute. If *handle2* is associated with an open file at the time of the call, that file is closed.

Warning: Both handles share a single file position. If you reposition a file using *handle1* , the position in the file returned by *handle2* will also change.

If you duplicate a file handle for an open stream (using fileno), the resulting file handle has the same restrictions as the original file handle.

Returns

dup2 returns `0` to indicate success. It returns `-1` if an error occurs and sets `errno` to one of the following values:

Value	**Meaning**
EBADF	The file handle is not valid.
EMFILE	No more file handles are available.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example makes a second file handle, `fh3`, refer to the same file as the file handle `fh1` using dup. The file handle `fh2` is then associated with the file `edopen.da2`, and finally `fh2` is forced to associate with `edopen.da1` by the dup2 function.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>

int main(void)
{
   int fh1,fh2,fh3;

   if (-1 == (fh1 = open("edopen.da1", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
      )) {
      perror("Unable to open edopen.da1");
      return EXIT_FAILURE;
   }
   if (-1 == (fh3 = dup(fh1))) {     /* fh3 refers to the sample file as fh1  */
      perror("Unable to dup");
      close(fh1);
      return EXIT_FAILURE;
   }
   else
      printf("Successfully performed dup handle.\n");
   if (-1 == (fh2 = open("edopen.da2", O_CREAT|O_TRUNC|O_RDWR, S_IREAD|S_IWRITE)
      )) {
      perror("Unable to open edopen.da2");
      close(fh1);
      close(fh3);
      return EXIT_FAILURE;
   }
   if (-1 == dup2(fh1, fh2)) {  /* Force fh2 to the refer to the same file     */
                                /* as fh1.                                    */
      perror("Unable to dup2");
   }
   else
      printf("Successfully performed dup2 handle.\n");
   close(fh1);
   close(fh2);
   close(fh3);
   return 0;

   /****************************************************************************
      The output should be:

       Successfully performed dup handle.
       Successfully performed dup2 handle.
   ****************************************************************************/
}
```

Related Information

- close
- creat
- dup
- open
- _sopen

_ecvt - Convert Floating-Point to Character

_ecvt - Convert Floating-Point to Character

Syntax

```
#include <stdlib.h>
char *_ecvt(double value, int ndigits, int *decptr, int *signptr);
```

Description

_ecvt converts the floating-point number *value* to a character string. _ecvt stores *ndigits* digits of *value* as a string and adds a null character (\0). If the number of digits in *value* exceeds *ndigits*, the low-order digit is rounded. If there are fewer than *ndigits* digits, the string is padded with zeros. Only digits are stored in the string.

You can obtain the position of the decimal point and the sign of *value* after the call from *decptr* and *signptr*. *decptr* points to an integer value that gives the position of the decimal point with respect to the beginning of the string. A 0 or a negative integer value indicates that the decimal point lies to the left of the first digit.

signptr points to an integer that indicates the sign of the converted number. If the integer value is 0, the number is positive. If it is not 0, the number is negative.

_ecvt also converts NaN and infinity values to the strings NAN and INFINITY, respectively. For more information on NaN and infinity values, see Infinity and NaN Support.

Warning: For each thread, the _ecvt, _fcvt and _gcvt functions use a single, dynamically allocated buffer for the conversion. Any subsequent call that the same thread makes to these functions destroys the result of the previous call.

Returns

_ecvt returns a pointer to the string of digits. Because of the limited precision of the double type, no more than 16 decimal digits are significant in any conversion. If it cannot allocate memory to perform the conversion. _ecvt returns `NULL` and sets errno to ENOMEM.

Example Code

This example reads in two floating-point numbers, calculates their product, and prints out only the billions digit of its character representation. At most, 16 decimal digits of significance can be expected. The output assumes the user enters the numbers 1000000 and 3000.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
   float x,y;
   double z;
   int w,b,decimal,sign;
   char *buffer;

   printf("Enter two floating-point numbers:\n");
   if (2 != scanf("%e %e", &x, &y)) {
      printf("input error...\n");
      return EXIT_FAILURE;
   }
   z = x *y;
   printf("Their product is %g\n", z);
   w = log10(fabs(z))+1.;
   buffer = _ecvt(z, w, &decimal, &sign);
   b = decimal-10;
   if (b < 0)
      printf("Their product does not exceed one billion.\n");
   else
      if (b > 15)
         printf("The billions digit of their product is insignificant.\n");
      else
         printf("The billions digit of their product is %c.\n", buffer[b]);
   return 0;

   /****************************************************************************
      For the following input:

      1000000 3000

      The output should be:

      Enter two floating-point numbers:
      1000000 3000
      Their product is 3e+09
      The billions digit of their product is 3.
   ****************************************************************************/
}
```

Related Information

- _fcvt
- _gcvt
- Infinity and NaN Support

_endthread - Terminate Current Thread

_endthread - Terminate Current Thread

Syntax

```
#include <stdlib.h>  /* also in <process.h> */
void _endthread(void);
```

Description

_endthread ends a thread that you previously created with _beginthread. When the thread has finished, it automatically ends itself with an implicit call to _endthread. You can also call _endthread explicitly to end the thread before it has completed its function, for example, if some error condition occurs.

Note: If you use DosCreateThread, you must explicitly call _endthread to terminate the thread.

Returns

There is no return value.

Example Code

In this example, the `main` program creates two threads, `bonjour` and `au_revoir`. The thread `bonjour` is forcibly terminated by a call to _endthread, while the `au_revoir` thread ends itself with an implicit call to _endthread.

```
#define  INCL_DOS
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

void bonjour(void *arg)
{
   int i = 0;

   while (i++ < 5)
      printf("Bonjour!\n");
   _endthread();                         /* This thread ends itself explicitly*/
   puts("thread should terminate before printing this");
}

void au_revoir(void *arg)
{
   int i = 0;

   while (i++ < 5)                      /* This thread makes an implicit      */
      printf("Au revoir!\n");                     /* call to _endthread       */
}

int main(void)
{
   unsigned long tid1;
   unsigned long tid2;

   tid1 = _beginthread(bonjour, NULL, 8192, NULL);
   tid2 = _beginthread(au_revoir, NULL, 8192, NULL);
   if (-1 == tid1 || -1 == tid2) {
      printf("Unable to start threads.\n");
      return EXIT_FAILURE;
   }
   DosWaitThread(&tid2, DCWW_WAIT);            /* wait until threads 1 and 2  */
   DosWaitThread(&tid1, DCWW_WAIT);             /* have been completed        */
   return 0;

   /****************************************************************************
      The output should be similar to:

      Au revoir!
      Au revoir!
      Au revoir!
      Au revoir!
      Au revoir!
      Bonjour!
      Bonjour!
      Bonjour!
      Bonjour!
      Bonjour!
   ****************************************************************************/
}
```

Related Information

- _beginthread

__eof - Determine End of File

__eof - Determine End of File

Syntax

```
#include <io.h>
int __eof (int handle);
```

Description

__eof determines whether the file pointer has reached the end-of-file for the file associated with *handle* . You cannot use __eof on a nonseekable file; it will fail.

Returns

__eof returns the value 1 if the current position is the end of the file or `0` if it is not. A return value of `-1` shows an error, and the system sets `errno` to the following values:

Value	**Meaning**
EBADF	File handle is not valid.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example creates the file `sample.dat` and then checks if the file pointer is at the end of that file using the `__eof` function.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>;

int main(void)
{
   int fh,returnValue;

   fh = creat("sample.dat", S_IREAD|S_IWRITE);
   if (-1 == fh) {
      perror("Error creating sample.dat");
      return EXIT_FAILURE;
   }
   if (-1 == (returnValue = __eof(fh))) {
      perror("eof function error");
      return EXIT_FAILURE;
   }
   if (1 == returnValue)
      printf("File pointer is at end-of-file position.\n");
   else
      printf("File pointer is not at end-of-file position.\n");
   close(fh);
   return 0;

   /****************************************************************************
      The output should be:

      File pointer is at end-of-file position.
   ****************************************************************************/
}
```

Related Information

- chsize
- creat
- _filelength
- _sopen
- open

----

erf - erfc - Calculate Error Functions

erf - erfc - Calculate Error Functions

Syntax

```
#include <math.h>
double erf(double x);
double erfc(double x);
```

Description

`erf` calculates the error function of

$$2\pi^{-1/2}\int_0^x e^{-t^2}\,dt$$

`erfc` computes the value of `1.0 - erf(`x`)`. `erfc` is used in place of `erf` for large values of x.

Returns

erf returns a double value that represents the error function. erfc returns a double value representing `1.0 - erf`.

Example Code

This example uses erf and erfc to compute the error function of two numbers.

```
#include <stdio.h>
#include <math.h>

double smallx,largex,value;

int main(void)
{
   smallx = 0.1;
   largex = 10.0;
   value = erf(smallx);                                  /* value = 0.112463 */
   printf("Error value for 0.1: %lf\n", value);
   value = erfc(largex);                         /* value = 2.088488e-45     */
   printf("Error value for 10.0: %le\n", value);
   return 0;
   /**************************************************************************
      The output should be:

      Error value for 0.1: 0.112463
      Error value for 10.0: 2.088488e-45
   **************************************************************************/
}
```

Related Information

- bessel
- gamma

----

execl - _execvpe - Load and Run Child Process

execl - _execvpe - Load and Run Child Process

Syntax

```
#include <process.h>
int execl(char *pathname, char *arg0, char *arg1,...,
             char *argn, NULL);
int execle(char *pathname, char *arg0, char *arg1,...,
             char *argn, NULL, char *envp[ ]);
int execlp(char *pathname, char *arg0, char *arg1,...,
             char *argn, NULL);
int _execlpe(char *pathname, char *arg0, char *arg1,...,
             char *argn, NULL, char *envp[ ]);
int execv(char *pathname, char *argv[ ]);
int execve(char *pathname, char *argv[ ],char *envp[ ]);
int execvp(char *pathname, char *argv[ ]);
int _execvpe(char *pathname, char *argv[ ], char *envp[ ]);
```

Description

The exec functions load and run new child processes. The parent process is ended after the child process has started. Sufficient storage must be available for loading and running the child process.

All of the exec functions are versions of the same routine; the letters at the end determine the specific variation:
compact break=fit.

Letter	Variation
p	Uses PATH environment variable to find the file to be run.
l	Passes a list of command line arguments separately.
v	Passes to the child process an array of pointers to command-line arguments.
e	Passes to the child process an array of pointers to environment strings.

Note: In earlier releases of C Set ++, all of the exec functions began with an underscore (`_getpid`). Because they are defined by the X/Open standard, the underscore has been removed. _execlpe and _execvpe retain the initial underscore because they are not included in the X/Open standard. For compatibility, *The Developer's Toolkit* will map the `_exec` functions to the correct exec function.

The *pathname* argument specifies the file to run as the child process. The *pathname* can specify a full path from the root, a partial path from the current working directory, or a file name. If *pathname* does not have a file name extension or does not end with a period, the exec functions will add the .EXE extension and search for the file. If *pathname* has an extension, the exec function uses only that extension. If *pathname* ends with a period, the exec functions search for *pathname* with no extension. The execlp, _execlpe, execvp, and _execvpe functions search for the *pathname* in the directories that the PATH environment variable specifies.

You pass arguments to the new process by giving one or more pointers to character strings as arguments in the exec call. These character strings form the argument list for the child process.

The compiler can pass the argument pointers as separate arguments (execl, execle, execlp, and _execlpe) or as an array of pointers (execv, execve, execvp, and _execvpe). You should pass at least one argument, either *arg0* or *argv* `[0]`, to the child process. If you do not, an argument will be returned that points to the same file as the path name argument you specified. This argument may not be exactly identical to the path name argument you specified. A different value does not produce an error.

Use the execl, execle, execlp, and _execlpe functions for the cases where you know the number of arguments in advance. The *arg0* argument is usually a pointer to *pathname* . The *arg1* through *argn* arguments are pointers to the character strings forming the new argument list. There must be a `NULL` pointer following `argn` to mark the end of the argument list.

Use the execv, execve, execvp, and _execvpe functions when the number of arguments to the new process is variable. Pass pointers to the arguments of these functions as an array, `argv[ ]`. The `argv[0]` argument is usually a pointer to *pathname* . The `argv[1]` through `argv[`*n*`]` arguments are pointers to the character strings forming the new argument list. If `argv`[*n*] is the last parameter, then `argv`[*n+1*] must be `NULL`.

Files that are open when you make an exec call remain open in the new process. In the execl, execlp, execv, and execvp calls, the child process receives the environment of the parent. The execle, _execlpe, execve, and _execvpe functions let you change the environment for the child process by passing a list of environment settings through the `envp` argument. The **envp** argument is an array of character pointers, each element of which points to a string ending with a null character that defines an environment variable. Such a string usually has the following form:

```
   NAME=value
```

where *NAME* is the name of an environment variable, and *value* is the string value to which the exec function sets that variable.

Note: Do not enclose the *value* in double quotation marks.

The final element of the **envp** array should be NULL. When **envp** itself is NULL, the child process receives the environment settings of the parent process.

The exec functions do not preserve signal settings in child processes created by calls to exec functions. Calls to exec functions reset the signal settings to the default in the child process.

Returns

The exec functions do not normally return control to the calling process. They are equivalent to the corresponding _spawn functions with P_OVERLAY as the value of *modeflag* . If an error occurs, the exec functions return `-1` and set errno to one of the following values: compact break=fit.

Value	**Meaning**
EACCESS	The specified file has a locking or sharing violation.
EMFILE	There are too many open files. The system must open the specified file to tell whether it is an executable file.
ENOENT	The file or *pathname* was not found or was specified incorrectly.
ENOEXEC	The specified file cannot run or has an incorrect executable file format.
ENOMEM	One of the following conditions exists: • Not enough storage is available to run the child process. • Not enough storage is available for the argument or environment strings.

Example Code

This example calls four of the eight exec routines. When invoked without arguments, the program first runs the code for case PARENT. It then calls execle() to load and run a copy of itself. The instructions for the child are blocked to run only if `argv[0]` and one parameter were passed (case CHILD). In its turn, the child runs its own child as a copy of the same program. This sequence is continued until four generations of child processes have run. Each of the processes prints a message identifying itself.

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

#define  PARENT       1
#define  CHILD        2

char *args[3];

int main(int argc, char **argv, char **envp) {
   switch(argc) {
      case PARENT: {                   /* No argument: run a child */
         printf("Parent process began.\n");
         execle(argv[0],argv[0],"1",NULL,envp);
         abort();     /* Not executed because parent was overlaid. */
      }



      case CHILD: {           /* One argument: run a child's child */
         printf("Child process %s began.\n", argv[1]);
         if ('1' == *argv[1]) {   /* generation one */
            execl(argv[0], argv[0], "2", NULL);
            abort();    /* Not executed because child was overlaid */
        }
        if('2' == *argv[1]) {                    /* generation two */
            args[0] = argv[0];
            args[1] = "3";
            args[2] = NULL;
            execv(argv[0],args);
            abort();    /* Not executed because child was overlaid */
         }
         if ('3' == *argv[1]) {                /* generation three */
            args[0] = argv[0];
            args[1] = "4";
            args[2] = NULL;
            execve(argv[0], args, _environ);
            abort();    /* Not executed because child was overlaid */
        }
         if ('4' == *argv[1])                   /* generation four */
            printf("Child process %s", argv[1]);
      }
   }
   printf(" ended.\n");
   return 55;
   /* The output should be similar to:
      Parent process began.
      Child process 1 began.
      Child process 2 began.
      Child process 3 began.
      Child process 4 began.
      Child process 4 ended.                                         */
}
```

Related Information

- abort
- _cwait
- exit
- _exit
- _spawnl - _spawnvpe
- system
- wait
- "envp Parameter to main" in the *VisualAge C++ Programming Guide*

----

exit - End Program

exit - End Program

Syntax

```
#include <stdlib.h>  /* also in <process.h> */
void exit(int status);
```

Description

`exit` returns control to the host environment from the program. It first calls all functions registered with the `atexit` function, in reverse order; that is, the last one registered is the first one called. See atexit for more information. It flushes all buffers and closes all open files before ending the program. All files opened with `tmpfile` are deleted.

The argument *status* can have a value from 0 to 255 inclusive or be one of the macros `EXIT_SUCCESS` or `EXIT_FAILURE`. A *status* value of `EXIT_SUCCESS` or `0` indicates a normal exit; otherwise, another status value is returned.

Returns

exit returns both control and the value of *status* to the operating system.

Example Code

This example ends the program after flushing buffers and closing any open files if it cannot open the file `myfile.mjq`.

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main(void)
{
   if (NULL == (stream = fopen("myfile.mjq", "r"))) {
      perror("Could not open data file");
      exit(EXIT_FAILURE);
   }
   return 0;

   /****************************************************************************
      The output should be:

      Could not open data file: The file cannot be found.
   ****************************************************************************/
}
```

Related Information

- abort
- atexit
- _onexit
- _exit
- signal
- tmpfile

_exit - End Process

_exit - End Process

Syntax

```
#include <stdlib.h>  /* also in <process.h> */
void _exit(int status);
```

Description

_exit ends the calling process without calling functions registered by _onexit or atexit. It also does not flush stream buffers or delete temporary files. You supply the *status* value as a parameter; the value `0` typically means a normal exit.

Returns

Although _exit does not return a value, the value is available to the waiting parent process, if there is one, after the child process ends. If no parent process waits for the exiting process, the *status* value is lost. The *status* value is available through the operating system batch command IF ERRORLEVEL.

Example Code

This example calls _exit to end the process. Because _exit does not flush the buffer first, the output from the second `printf` statement will not appear.

```
#include <stdio.h>
#include <stdlib.h>                      /* You can also use <process.h>      */

char buf[51];

int main(void)
{

   /* Make sure the standard output stream is line-buffered even if the       */
   /* output is redirected to a file.                                         */

   if (0 != setvbuf(stdout, buf, _IOLBF, 50))
      printf("The buffering was not set correctly.\n");
   printf("This will print out but ...\n");
   printf("this will not!");
   _exit(EXIT_FAILURE);
   return 0;

   /****************************************************************************
      The output should be:

      This will print out but ...
   ****************************************************************************/
}
```

Related Information

- abort
- atexit
- execl - _execvpe
- exit
- _onexit

----

exp - Calculate Exponential Function

exp - Calculate Exponential Function

Syntax

```
#include <math.h>
double exp(double x);
```

Description

`exp` calculates the exponential function of a floating-point argument *x* (*ex*, where *e* equals 2.17128128...).

Returns

If an overflow occurs, exp returns `HUGE_VAL`. If an underflow occurs, it returns `0`. Both overflow and underflow set `errno` to `ERANGE`.

Example Code

This example calculates *y* as the exponential function of *x*:

```
#include <math.h>

int main(void)
{
   double x,y;

   x = 5.0;
   y = exp(x);
   printf("exp( %lf ) = %lf\n", x, y);
   return 0;

   /****************************************************************************
      The output should be:

      exp( 5.000000 ) = 148.413159
   ****************************************************************************/
}
```

Related Information

- log
- log10

----

fabs - Calculate Floating-Point Absolute Value

fabs - Calculate Floating-Point Absolute Value

Syntax

```
#include <math.h>
double fabs(double x);
```

Description

`fabs` calculates the absolute value of the floating-point argument *x*.

Returns

fabs returns the absolute value. There is no error return value.

Example Code

This example calculates *y* as the absolute value of *x*:

```
#include <math.h>
```

```
int main(void)
{
   double x,y;

   x = -5.6798;
   y = fabs(x);
   printf("fabs( %lf ) = %lf\n", x, y);
   return 0;

   /*****************************************************************************
      The output should be similar to :

      fabs( -5.679800 ) = 5.679800
   *****************************************************************************/
}
```

Related Information

- abs
- labs

----

fclose - Close Stream

fclose - Close Stream

Syntax

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

`fclose` closes a stream pointed to by *stream* . This function flushes all buffers associated with the stream before closing it. When it closes the stream, the function releases any buffers that the system reserved. When a binary stream is closed, the last record in the file is padded with null characters (`\0`) to the end of the record.

Returns

`fclose` returns `0` if it successfully closes the stream, or `EOF` if any errors were detected.

Note: Once you close a stream with `fclose`, you must open it again before you can use it.

Example Code

This example opens a file `fclose.dat` for reading as a stream; then it closes this file.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   stream = fopen("fclose.dat", "r");
   if (0 != fclose(stream))                              /* Close the stream */
       perror("fclose error");
   else
      printf("File closed successfully.\n");
   return 0;

   /*****************************************************************************
      The output should be:
```

```
      File closed successfully.
   ****************************************************************************/
}
```

Related Information

- close
- _fcloseall
- fflush
- fopen
- freopen

_fcloseall - Close All Open Streams

_fcloseall - Close All Open Streams

Syntax

```
#include <stdio.h>
int _fcloseall(void);
```

Description

_fcloseall closes all open streams, except `stdin`, `stdout`, and `stderr`. It also closes and deletes any temporary files created by `tmpfile`.

_fcloseall flushes all buffers associated with the streams before closing them. When it closes streams, it releases the buffers that the system reserved, but does not release user-allocated buffers.

Returns

_fcloseall returns the total number of streams closed, or EOF if an error occurs.

Example Code

This example opens a file `john` for reading as a data stream, and then closes the file. It closes all other streams except `stdin`, `stdout`, and `stderr`.

```
#include <stdio.h>

#define OUTFILE "temp.out"

int main(void)
{
   FILE *stream;
   int numclosed;

   stream = fopen(OUTFILE, "w");
   numclosed = _fcloseall();
   printf("Number of files closed = %d\n", numclosed);
   remove(OUTFILE);
   return 0;

   /****************************************************************************
      The output should be:

      Number of files closed = 1
   ****************************************************************************/
}
```

Related Information

- close
- fclose
- fflush
- fopen
- freopen
- tmpfile

_fcvt - Convert Floating-Point to String

_fcvt - Convert Floating-Point to String

Syntax

```
#include <stdlib.h>
char *_fcvt(double value, int ndec, int *decptr, int *signptr);
```

Description

_fcvt converts the floating-point number *value* to a character string. _fcvt stores the digits of *value* as a string and adds a null character (\0). The *ndec* variable specifies the number of digits to be stored after the decimal point.

If the number of digits after the decimal point in *value* exceeds *ndec* , _fcvt rounds the correct digit according to the FORTRAN F format. If there are fewer than *ndec* digits of precision, _fcvt pads the string with zeros.

A FORTRAN F number has the following format:

```
>>             digit    .                 ><
       +                      digit
```

_fcvt stores only digits in the string. You can obtain the position of the decimal point and the sign of *value* after the call from *decptr* and *signptr* . *decptr* points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value shows that the decimal point lies to the left of the first digit.

signptr points to an integer showing the sign of *value* . _fcvt sets the integer to 0 if *value* is positive and to a nonzero number if *value* is negative.

_fcvt also converts NaN and infinity values to the strings NAN and INFINITY, respectively. For more information on NaN and infinity values, see Infinity and NaN Support.

Warning: For each thread, the _ecvt, _fcvt, and _gcvt functions use a single, dynamically allocated buffer for the conversion. Any subsequent call that the same thread makes to these functions destroys the result of the previous call.

Returns

_fcvt returns a pointer to the string of digits. Because of the limited precision of the double type, no more than 16 decimal digits are significant in any conversion. If it cannot allocate memory to perform the conversion, _fcvt returns NULL and sets errno to ENOMEM.

Example Code

This example reads in two floating-point numbers, computes their product, and prints out only the billions digit of its character representation. At most, 16 decimal digits of significance can be expected. The output given assumes the user enters the values 2000000 and 3000.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
   float x = 2000000;
   float y = 3000;
   double z;
   int w,b,decimal,sign;
   char *buffer;

   z = x *y;
   printf("The product of %e and %e is %g.\n", x, y, z);
   w = log10(fabs(z))+1.;
   buffer = _fcvt(z, w, &decimal, &sign);
   b = decimal-10;
   if (b < 0)
      printf("Their product does not exceed one billion.\n");
   if (b > 15)
      printf("The billions digit of their product is insignificant.\n");
   if ((b > -1) && (b < 16))
      printf("The billions digit of their product is %c.\n", buffer[b]);
   return 0;

   /****************************************************************************
      The output should be:

      The product of 2.000000e+06 and 3.000000e+03 is 6e+09.
      The billions digit of their product is 6.
   ****************************************************************************/
}
```

Related Information

- _ecvt
- _gcvt
- Infinity and NaN Support

----

fdopen - Associates Input Or Output With File

fdopen - Associates Input Or Output With File

Syntax

```
#include <stdio.h>
FILE *fdopen(int handle, char *type);
```

Description

fdopen associates an input or output stream with the file identified by *handle* . The *type* variable is a character string specifying the type of access requested for the stream.

Mode	Description
r	Create a stream to read a text file. The file pointer is set to the beginning of the file.
w	Create a stream to write to a text file. The file pointer is set to the beginning of the file.

`a`	Create a stream to write, in append mode, at the end of the text file. The file pointer is set to the end of the file.
`r+`	Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.
`w+`	Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.
`a+`	Create a stream for reading or writing, in append mode, at the end of the text file. The file pointer is set to the end of the file.
`rb`	Create a stream to read a binary file. The file pointer is set to the beginning of the file.
`wb`	Create a stream to write to a binary file. The file pointer is set to the beginning of the file.
`ab`	Create a stream to write to a binary file in append mode. The file pointer is set to the end of the file.
`r+b` *or* `rb+`	Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.
`w+b` *or* `wb+`	Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.
`a+b` *or* `ab+`	Create a stream for reading and writing to a binary file in append mode. The file pointer is set to the end of the file.

Warning: Use the `w, w+, wb, wb+,` and `w+b` modes with care; they can destroy existing files.

The specified *type* must be compatible with the access mode you used to open the file. If the file was opened with the O_APPEND FLAG, the stream mode must be `r, a, a+, rb, ab, a+b,` or `ab+`.

When you open a file with `a, a+, ab, a+b`, or `ab+` as the value of *type* , all write operations take place at the end of the file. Although you can reposition the file pointer using `fseek` or `rewind`, the file pointer always moves back to the end of the file before the system carries out any write operation. This action prevents you from writing over existing data.

When you specify any of the types containing `+`, you can read from and write to the file, and the file is open for update. However, when switching from reading to writing or from writing to reading, you must include an intervening `fseek`, `fsetpos`, or `rewind` operation. You can specify the current file position with `fseek`.

In accessing text files, carriage-return line-feed (CR-LF) combinations are translated into a single line feed (LF) on input; LF characters are translated to CR-LF combinations on output. Accesses to binary files suppress all of these translations. (See "Stream Processing" in the *VisualAge C++ Programming Guide* for the differences between text and binary streams.)

If fdopen returns `NULL`, use close to close the file. If fdopen is successful, you must use fclose to close the stream and file.

Returns

fdopen returns a pointer to a file structure that can be used to access the open file. A `NULL` pointer return value indicates an error.

Example Code

This example opens the file `sample.dat` and associates a stream with the file using `fdopen`. It then reads from the stream into the buffer.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
   long length;
   int fh;
   char buffer[20];
   FILE *fp;

   memset(buffer, '\0', 20);                              /* Initialize buffer*/
   printf("\nCreating sample.dat.\n");
   system("echo Sample Program > sample.dat");
   if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
      perror("Unable to open sample.dat");
      return EXIT_FAILURE;
   }
   if (NULL == (fp = fdopen(fh, "r"))) {
      perror("fdopen failed");
      close(fh);
      return EXIT_FAILURE;
   }
   if (7 != fread(buffer, 1, 7, fp)) {
      perror("fread failed");
      fclose(fp);
      return EXIT_FAILURE;
   }
   printf("Successfully read from the stream the following:\n%s.\n", buffer);
   fclose(fp);
   return 0;

   /****************************************************************************
      The output should be:

      Creating sample.dat.
      Successfully read from the stream the following:
      Sample .
   ****************************************************************************/
}
```

Related Information

- close
- creat
- fclose
- fopen
- fseek
- fsetpos
- open
- rewind
- _sopen

----

feof - Test End-of-File Indicator

feof - Test End-of-File Indicator

Syntax

```
#include <stdio.h>
int feof(FILE *stream);
```

Description

`feof` indicates whether the end-of-file flag is set for the given *stream* . The end-of-file flag is set by several functions to indicate the end of the file. The end-of-file flag is cleared by calling `rewind`, `fsetpos`, `fseek`, or `clearerr` for this stream.

Returns

`feof` returns a nonzero value if and only if the `EOF` flag is set; otherwise, it returns `0`.

Example Code

This example scans the input `stream` until it reads an end-of-file character.

```
#include <stdio.h>

int main(void)
{
   char inp_char;
   FILE *stream;

   stream = fopen("feof.dat", "r");

   /* scan an input stream until an end-of-file character is read             */

   while (0 == feof(stream)) {
      fscanf(stream, "%c", &inp_char);
      printf("<x%x> ", inp_char);
   }
   fclose(stream);
   return 0;

   /****************************************************************************
      If feof.dat contains : abc defgh

      The output should be:

      <x61> <x62> <x63> <x20> <x64> <x65> <x66> <x67> <x68>
   ****************************************************************************/
}
```

Related Information

- clearerr
- ferror
- fseek
- fsetpos
- perror
- rewind

----

ferror - Test for Read/Write Errors

ferror - Test for Read/Write Errors

Syntax

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

`ferror` tests for an error in reading from or writing to the given *stream* . If an error occurs, the error indicator for the *stream* remains set until you close *stream* , call `rewind`, or call `clearerr`.

Returns

The `ferror` function returns a nonzero value to indicate an error on the given *stream* . A return value of `0` means no error has occurred.

Example Code

This example puts data out to a stream and then checks that a write error has not occurred.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *stream;
   char *string = "Important information";

   stream = fopen("ferror.dat", "w");
   fprintf(stream, "%s\n", string);
   if (ferror(stream)) {
      printf("write error\n");
      clearerr(stream);
      return EXIT_FAILURE;
   }
   else
      printf("Data written to a file successfully.\n");
   if (fclose(stream))
      perror("fclose error");
   return 0;

   /*****************************************************************************
      The output should be:

      Data written to a file successfully.
   *****************************************************************************/
}
```

Related Information

- clearerr
- feof
- fopen
- perror
- strerror
- _strerror

----

fflush - Write Buffer to File

fflush - Write Buffer to File

Syntax

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

`fflush` causes the system to empty the buffer associated with the specified output *stream* , if possible. If the *stream* is open for input, `fflush` undoes the effect of any `ungetc` function. The *stream* remains open after the call.

If *stream* is `NULL`, the system flushes all open streams.

Note: The system automatically flushes buffers when you close the stream, or when a program ends normally without closing the stream.

Returns

`fflush` returns the value `0` if it successfully flushes the buffer. It returns `EOF` if an error occurs.

Example Code

This example flushes a stream buffer.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   stream = fopen("myfile.dat", "w");
   fprintf(stream, "Hello world");
   fflush(stream);
   fclose(stream);
   return 0;
}
```

Related Information

- fclose
- _flushall
- setbuf
- ungetc

----

fgetc - Read a Byte

fgetc - Read a Byte

Syntax

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

`fgetc` reads a single byte from the input *stream* at the current position and increases the associated file pointer, if any, so that it points to the next byte.

Note: `fgetc` is identical to `getc` but is always implemented as a function call; it is never replaced by a macro.

Returns

`fgetc` returns the byte read as an integer. An `EOF` return value indicates an error or an end-of-file condition. Use `feof` or `ferror` to determine whether the `EOF` value indicates an error or the end of the file.

Example Code

This example gathers a line of input from a stream.

```
#include <stdio.h>

#define  MAX_LEN       80

int main(void)
```

```
{
   FILE *stream;
   char buffer[MAX_LEN+1];
   int i,ch;

   stream = fopen("myfile.dat", "r");
   for (i = 0; (i < (sizeof(buffer)-1) && ((ch = fgetc(stream)) != EOF) &&
               (ch != '\n')); i++)
      buffer[i] = ch;
   buffer[i] = '\0';
   if (fclose(stream))
      perror("fclose error");
   printf("The input line was : %s\n", buffer);
   return 0;

   /****************************************************************************
      If myfile.dat contains: one two three

      The output should be:

      The input line was : one two three
   ****************************************************************************/
}
```

Related Information

- feof
- ferror
- fputc
- getc - getchar
- _getch - _getche

----

fgetpos - Get File Position

fgetpos - Get File Position

Syntax

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

`fgetpos` stores the current position of the file pointer associated with *stream* into the object pointed to by *pos* . The value pointed to by *pos* can be used later in a call to `fsetpos` to reposition the *stream* .

Note: For buffered text streams, fgetpos returns an incorrect file position if the file contains new-line characters instead of carriage-return line-feed combinations. Your file would only contain new-line characters if you previously used it as a binary stream. To avoid this problem, either continue to process the file as a binary stream, or use unbuffered I/O operations.

Returns

`fgetpos` returns `0` if successful. On error, `fgetpos` returns nonzero and sets `errno` to a nonzero value.

Example Code

This example opens the file myfile.dat for reading and stores the current file pointer position into the variable *pos* .

```
#include <stdio.h>
```

```
FILE *stream;

int main(void)
{
   int retcode;
   fpos_t pos;

   stream = fopen("myfile.dat", "rb");

    /* The value returned by fgetpos can be used by fsetpos                   */
    /* to set the file pointer if 'retcode' is 0                              */

   if ( 0 == (retcode = fgetpos(stream, &pos)) )
      printf("Current position of file pointer found.\n");
   fclose(stream);
   return 0;

   /****************************************************************************
      If myfile.dat exists, the output should be:

      Current position of file pointer found.
   ****************************************************************************/
}
```

Related Information

- fseek
- fsetpos
- ftell

----

fgets - Read a String

fgets - Read a String

Syntax

```
#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);
```

Description

`fgets` reads bytes from the current *stream* position up to and including the first new-line character (`\n`), up to the end of the stream, or until the number of bytes read is equal to *n*-1, whichever comes first. `fgets` stores the result in *string* and adds a null character (\`0`) to the end of the string. The *string* includes the new-line character, if read. If *n* is equal to 1, the *string* is empty.

Returns

`fgets` returns a pointer to the *string* buffer if successful. A `NULL` return value indicates an error or an end-of-file condition. Use `feof` or `ferror` to determine whether the `NULL` value indicates an error or the end of the file. In either case, the value of the string is unchanged.

Example Code

This example gets a line of input from a data stream. The example reads no more than `MAX_LEN - 1` characters, or up to a new-line character, from the stream.

```
#include <stdio.h>

#define  MAX_LEN       100
```

```
int main(void)
{
   FILE *stream;
   char line[MAX_LEN],*result;

   stream = fopen("myfile.dat", "rb");
   if ((result = fgets(line, MAX_LEN, stream)) != NULL)
      printf("The string is %s\n", result);
   if (fclose(stream))
      perror("fclose error");
   return 0;

   /*****************************************************************************
      If myfile.dat contains: This is my data file.

      The output should be:

      The string is This is my data file.
   *****************************************************************************/
}
```

Related Information

- feof
- ferror
- fputs
- gets
- puts

----

fgetwc - Read Wide Character from Stream

fgetwc - Read Wide Character from Stream

Syntax

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

fgetwc reads the next multibyte character from the input stream pointed to by *stream* , converts it to a wide character, and advances the associated file position indicator for the stream (if defined).

The behavior of fgetwc is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

After calling fgetwc, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling fgetwc.

Returns

fgetwc returns the next wide character that corresponds to the multibyte character from the input stream pointed to by *stream* . If the stream is at EOF, the EOF indicator for the stream is set and fgetwc returns WEOF.

If a read error occurs, the error indicator for the stream is set and fgetwc returns WEOF. If an encoding error occurs (an error converting the multibyte character into a wide character), fgetwc sets errno to EILSEQ and returns WEOF.

Use ferror and feof to distinguish between a read error and an EOF. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

Example Code

This example opens a file, reads in each wide character using fgetwc, and prints out the characters.

```
#include <stdio.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE   *stream;
   wint_t wc;

   if (NULL == (stream = fopen("fgetwc.dat", "r"))) {
      printf("Unable to open: \"fgetwc.dat\"\n");
      exit(1);
   }

   errno = 0;
   while (WEOF != (wc = fgetwc(stream)))
      printf("wc = %lc\n", wc);

   if (EILSEQ == errno) {
      printf("An invalid wide character was encountered.\n");
      exit(1);
   }
   fclose(stream);
   return 0;

   /****************************************************************************
      Assuming the file fgetwc.dat contains:

      Hello world!

      The output should be similar to:

      wc = H
      wc = e
      wc = l
      wc = l
      wc = o
      :
   ****************************************************************************/
}
```

Related Information

- fgetc
- fgetws
- fputwc
- _getch - _getche

----

fgetws - Read Wide-Character String from Stream

fgetws - Read Wide-Character String from Stream

Syntax

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);
```

Description

fgetws reads wide characters from the *stream* into the array pointed to by *wcs*. At most, *n* - 1 wide characters are read. fgetws stops reading characters after WEOF, or after it reads a new-line wide character (which is retained). It adds a null wide character immediately after the last wide character read into the array.

fgetws advances the file position unless there is an error. If an error occurs, the file position is undefined.

The behavior of fgetws is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

After calling fgetws, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless WEOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling fgetws.

Returns

If successful, fgetws returns a pointer to the wide-character string *wcs* . If WEOF is encountered before any wide characters have been read into *wcs* , the contents of *wcs* remain unchanged and fgetws returns a null pointer. If WEOF is reached after data has already been read into the string buffer, fgetws returns a pointer to the string buffer to indicate success. A subsequent call would return NULL because WEOF would be reached without any data being read.

If a read error occurs, the contents of *wcs* are indeterminate and fgetws returns NULL. If an encoding error occurs (in converting a wide character to a multibyte character), fgetws sets errno to EILSEQ and returns NULL.

If *n* equals 1, the *wcs* buffer has only room for the terminating null character and nothing is read from the stream. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless the buffer is flushed or the stream pointer repositioned first.)

If *n* is greater than 1, fgetws fails only if an I/O error occurs or if WEOF is reached before data is read from the stream. Use ferror and feof to distinguish between a read error and a WEOF. WEOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the WEOF indicator.

Example Code

This example opens a file, reads in the file contents using fgetws, then prints the file contents.

```
#include <errno.h>
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wchar_t  wcs[100];

   if (NULL == (stream = fopen("fgetws.dat", "r"))) {
      printf("Unable to open: \"fgetws.dat\"\n");
      exit(1);
   }

   errno = 0;
   if (NULL == fgetws(wcs, 100, stream)) {
      if (EILSEQ == errno) {
         printf("An invalid wide character was encountered.\n");
         exit(1);
      }
      else if (feof(stream))
              printf("End of file reached.\n");
           else
              perror("Read error.\n");
   }
   printf("wcs = \"%ls\"\n", wcs);
   fclose(stream);
   return 0;

   /****************************************************************************
      Assuming the file fgetws.dat contains:

      This test string should not return -1

      The output should be similar to:

      wcs = "This test string should not return -1"
   ****************************************************************************/
}
```

Related Information

- fgets
- fgetwc
- fputws.

_filelength - Determine File Length

_filelength - Determine File Length

Syntax

```
#include <io.h>
long _filelength(int handle);
```

Description

_filelength returns the length, in bytes, of the file associated with *handle* . The length of the file will be correct even if you have the handle opened and have appended data to the file.

Returns

A return value of `-1L` indicates an error, and `errno` is set to one of the following values: compact break=fit.

Value	**Meaning**
EBADF	The file handle is incorrect or the mode specified does not match the mode you opened the file with.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example opens a file and tries to determine the current length of the file using _filelength.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
   long length;
   int fh;

   printf("\nCreating sample.dat.\n");
   system("echo Sample Program > sample.dat");
   if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
      printf("Unable to open sample.dat.\n");
      return EXIT_FAILURE;
   }
   if (-1 == (length = _filelength(fh))) {
      printf("Unable to determine length of sample.dat.\n");
      return EXIT_FAILURE;
   }
   printf("Current length of sample.dat is %d.\n", length);
   close(fh);
   return 0;

   /****************************************************************************
      The output should be:

      Creating sample.dat.
      Current length of sample.dat is 17.
   ****************************************************************************/
}
```

Related Information

- chsize
- __eof

----

fileno - Determine File Handle

fileno - Determine File Handle

Syntax

```
#include <stdio.h>
int fileno(FILE *stream);
```

Description

fileno determines the file handle currently associated with *stream* .

Note: In earlier releases of C Set ++, fileno began with an underscore (`_fileno`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_fileno` to fileno for you.

Returns

fileno returns the file handle. If the function fails, the return value is -1 and the errno variable may be set to one of the following values: compact break=fit.

Value	Meaning
ENULLFCB	The input stream is `NULL`.
EBADTYPE	The input stream file is not a stream file.

The result is undefined if *stream* does not specify an open file.

Example Code

This example determines the file handle of the stderr data stream.

```
#include <stdio.h>

int main(void)
{
   int result;

   result = 0xFFFF & fileno(stderr);
   printf("The file handle associated with stderr is %d.\n", result);
   return 0;

   /****************************************************************************
      The output should be:

      The file handle associated with stderr is 2.
   ****************************************************************************/
}
```

Related Information

- dup
- dup2
- fopen
- freopen
- isatty
- open

---

floor - Integer

floor - Integer <= Argument

Syntax

```
#include <math.h>
double floor(double x);
```

Description

`floor` calculates the largest integer that is less than or equal to *x*.

Returns

`floor` returns the floating-point result as a double value.

The result of `floor` cannot have a range error.

Example Code

This example assigns *y* value of the largest integer less than or equal to 2.8 and *z* the value of the largest integer less than or equal to -2.8.

```
#include <math.h>

int main(void)
{
   double y,z;

   y = floor(2.8);
   z = floor(-2.8);
   printf("floor(  2.8 ) = %lf\n", y);
   printf("floor( -2.8 ) = %lf\n", z);
   return 0;

   /****************************************************************************
      The output should be:

      floor(  2.8 ) = 2.000000
      floor( -2.8 ) = -3.000000
   ****************************************************************************/
}
```

Related Information

- ceil
- fmod

_flushall - Write Buffers to Files

_flushall - Write Buffers to Files

Syntax

```
#include <stdio.h>
int _flushall(void);
```

Description

_flushall causes the system to write to file the contents of all buffers associated with open output streams (including stdin, stdout, and stderr). It clears all buffers associated with open input streams of their current contents. The next read operation, if there is one, reads new data from the input files into the buffers. All streams remain open after the call.

For portability, use the ANSI/ISO function `fflush` instead of _flushall.

Returns

_flushall returns the number of open streams of input and output. If an error occurs, _flushall returns `EOF`.

Example Code

In this example, _flushall completes any pending input or output on all streams by flushing all buffers.

```
#include <stdio.h>

int main(void)
{
   int i,numflushed;
   char buffer1[5] =  { 1,2,3,4 };
   char buffer2[5] =  { 5,6,7,8 };
   char *file1 = "file1.dat";
   char *file2 = "file2.dat";
   FILE *stream1,*stream2;

   stream1 = fopen(file1, "a+");
   stream2 = fopen(file2, "a+");
   for (i = 0; i <= sizeof(buffer1); i++) {
      fputc(buffer1[i], stream1);
      fputc(buffer2[i], stream2);
   }
   numflushed = _flushall();                   /* all streams flushed          */
   printf("Number of files flushed = %d\n", numflushed);
   fclose(stream1);
   fclose(stream2);
   return 0;

   /****************************************************************************
      The output should be:

      Number of files flushed = 5
   ****************************************************************************/
}
```

Related Information

- close
- fclose
- fflush

----

fmod - Calculate Floating-Point Remainder

fmod - Calculate Floating-Point Remainder

Syntax

```
#include <math.h>
double fmod(double x, double y);
```

Description

`fmod` calculates the floating-point remainder of *x*/*y*. The absolute value of the result is always less than the absolute value of *y*. The result will have the same sign as *x*.

Returns

`fmod` returns the floating-point remainder of *x*/*y*. If *y* is zero or if *x*/*y* causes an overflow, fmod returns `0`.

Example Code

This example computes *z* as the remainder of *x*/*y*; here, *x*/*y* is -3 with a remainder of -1.

```
#include <math.h>

int main(void)
{
   double x,y,z;

   x = -10.0;
   y = 3.0;
   z = fmod(x, y);                                  /* z = -1.0         */
   printf("fmod( %lf, %lf) = %lf\n", x, y, z);
   return 0;

   /****************************************************************************
      The output should be:

      fmod( -10.000000, 3.000000) = -1.000000
   ****************************************************************************/
}
```

Related Information

- ceil
- fabs
- floor

----

fopen - Open Files

fopen - Open Files

Syntax

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Description

`fopen` opens the file specified by *filename* . *mode* is a character string specifying the type of access requested for the file. The *mode* variable contains one positional parameter followed by optional keyword parameters.

The possible values for the positional parameters are:

Mode	Description
`r`	Open a text file for reading. The file must exist.
`w`	Create a text file for writing. If the given file exists, its contents are destroyed.
`a`	Open a text file in append mode for writing at the end of the file. `fopen` creates the file if it does not exist.
`r+`	Open a text file for both reading and writing. The file must exist.
`w+`	Create a text file for both reading and writing. If the given file exists, its contents are destroyed.
`a+`	Open a text file in append mode for reading or updating at the end of the file. `fopen` creates the file if it does not exist.
`rb`	Open a binary file for reading. The file must exist.
`wb`	Create an empty binary file for writing. If the file exists, its contents are destroyed.
`ab`	Open a binary file in append mode for writing at the end of the file. `fopen` creates the file if it does not exist.
`r+b` *or* `rb+`	Open a binary file for both reading and writing. The file must exist.
`w+b` *or* `wb+`	Create an empty binary file for both reading and writing. If the file exists, its contents will be destroyed.
`a+b` *or* `ab+`	Open a binary file in append mode for writing at the end of the file. `fopen` creates the file if it does not exist.

Warning: Use the `w`, `w+`, `wb`, `w+b`, and `wb+` parameters with care; data in existing files of the same name will be lost.

Text files contain printable characters and control characters organized into lines. Each line ends with a new-line character, except for the last line, which does not require one. The system may insert or convert control characters in an output text stream.

Note: Data output to a text stream may not compare as equal to the same data on input.

Binary files contain a series of characters. For binary files, the system does not translate control characters on input or output.

When you open a file with `a`, `a+`, `ab`, `a+b` or `ab+` mode, all write operations take place at the end of the file. Although you can reposition the file pointer using `fseek` or `rewind`, the write functions move the file pointer back to the end of the file before they carry out any operation. This action prevents you from overwriting existing data.

When you specify the update mode (using `+` in the second or third position), you can both read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as `fseek`, `fsetpos`, `rewind`, or `fflush`. Output may immediately follow input if the end-of-file was detected.

The keyword parameters are:

`blksize=`*value*

Specifies the maximum length, in bytes, of a physical block of records. For fixed-length records, the maximum size is 32760 bytes. For variable-length records, the maximum is 32756. The default buffer size is 4096 bytes.

`lrecl=`*value*

Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records. For fixed-length records, the maximum length is 32760 bytes. For variable-length records, the maximum is 32756. If the value of LRECL is larger than the value of BLKSIZE, the LRECL value is ignored.

`recfm=`*value*

value can be:

`F`	fixed-length, unblocked records
`V`	variable-length, unblocked records The default for *The Developer's Toolkit* compiler is fixed-length record format.

`type=`*value*

value can be:

memory This parameter identifies this file as a memory file that is accessible only from C programs. This is the default.

The Developer's Toolkit compiler does not support record I/O.

`fopen` generally fails if parameters are mismatched.

The file attributes can be altered only if the open mode specified with the `fopen` function is one of the write modes (`w, w+, wb`, or `wb+`). The system deletes the existing file and creates a new file with the attributes specified in `fopen`.

Returns

`fopen` returns a pointer to a file structure that can be used to access the open file. A `NULL` pointer return value indicates an error.

Example Code

This example attempts to open a file for reading.

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   if (NULL == (stream = fopen("myfile.dat", "r")))
      printf("Could not open data file\n");
   else
      fclose(stream);

    /* The following call opens a fixed record length file                    */
    /* for reading and writing in record mode.                                */

   if (NULL == (stream = fopen("myfile.dat",
                     "rb+, lrecl=80, blksize=240, recfm=f, type=record")))
      printf("Could not open data file\n");
   else
      fclose(stream);
   return 0;

   /****************************************************************************
      The output should be:

      Could not open data file
   ****************************************************************************/
}
```

Related Information

- creat
- fclose
- fflush
- fread
- freopen
- open
- _sopen
- fseek
- fsetpos
- fwrite
- rewind

----

fprintf - Write Formatted Data to a Stream

fprintf - Write Formatted Data to a Stream

Syntax

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format-string, argument-list);
```

Description

`fprintf` formats and writes a series of characters and values to the output *stream* . `fprintf` converts each entry in *argument-list* , if any, and writes to the stream according to the corresponding format specification in the *format-string* .

The *format-string* has the same form and function as the *format-string* argument for `printf`. See printf for a description of the *format-string* and the argument list.

In extended mode, `fprintf` also converts floating-point values of NaN and infinity to the strings `"NAN"` or `"nan"` and `"INFINITY"` or `"infinity"`. The case and sign of the string is determined by the format specifiers. See Infinity and NaN Support for more information on infinity and NaN values.

Returns

`fprintf` returns the number of bytes printed or a negative value if an output error occurs.

Example Code

This example sends a line of asterisks for each integer in the array `count` to the file `myfile`. The number of asterisks printed on each line corresponds to an integer in the array.

```
#include <stdio.h>

int count[10] =  { 1, 5, 8, 3, 0, 3, 5, 6, 8, 10 } ;

int main(void)
{
   int i,j;
   FILE *stream;

   stream = fopen("myfile.dat", "w");

                   /* Open the stream for writing                             */

   for (i = 0; i < sizeof(count)/sizeof(count[0]); i++) {
      for (j = 0; j < count[i]; j++)
         fprintf(stream, "*");

                   /* Print asterisk                                          */

      fprintf(stream, "\n");

                   /* Move to the next line                                   */

   }
   fclose(stream);
   return 0;

   /****************************************************************************
      The output data file myfile.dat should contain:

      *
      *****
      ********
      ***

      ***
      *****
      ******
      ********
      **********
   ****************************************************************************/
}
```

Related Information

- _cprintf
- fscanf
- printf
- sprintf
- vfprintf
- vprintf
- vsprintf
- Infinity and NaN Support

----

fputc - Write a Byte

fputc - Write a Byte

Syntax

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description

`fputc` converts *c* to an `unsigned char` and then writes *c* to the output *stream* at the current position and advances the file position appropriately. If the stream is opened with one of the append modes, the character is appended to the end of the stream.

`fputc` is identical to `putc` is always implemented as a function call; it is never replaced by a macro.

Returns

`fputc` returns the byte written. A return value of `EOF` indicates an error.

Example Code

This example writes the contents of `buffer` to a file called myfile.dat.

Note: Because the output occurs as a side effect within the second expression of the `for` statement, the statement body is null.

```
#include <stdio.h>
#include <stdlib.h>

#define  NUM_ALPHA     26

int main(void)
{
   FILE *stream;
   int i;
   int ch;

   /* Don't forget the NULL char at the end of the string!                    */

   char buffer[NUM_ALPHA+1] = "abcdefghijklmnopqrstuvwxyz";

   if ((stream = fopen("myfile.dat", "w")) != NULL) {

     /* Put buffer into file                                                  */

      for (i = 0; (i < sizeof(buffer)) && ((ch = fputc(buffer[i], stream)) !=
         EOF); ++i)
         ;
      fclose(stream);
      return 0;
   }
   else
      perror("Error opening myfile.dat");
   return EXIT_FAILURE;

   /****************************************************************************
      The output data file myfile.dat should contain:

      abcdefghijklmnopqrstuvwxyz
   ****************************************************************************/
}
```

Related Information

- fgetc
- putc - putchar
- _putch

----

fputs - Write String

fputs - Write String

Syntax

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

Description

`fputs` copies *string* to the output *stream* at the current position. It does not copy the null character (`\0`) at the end of the string.

Returns

fputs returns `EOF` if an error occurs; otherwise, it returns a non-negative value.

Example Code

This example writes a string to a stream.

```
#include <stdio.h>

#define  NUM_ALPHA     26

int main(void)
{
   FILE *stream;
   int num;

   /* Do not forget that the '\0' char occupies one character                 */

   static char buffer[NUM_ALPHA+1] = "abcdefghijklmnopqrstuvwxyz";

   if ((stream = fopen("myfile.dat", "w")) != NULL) {

      /* Put buffer into file                                                 */

      if ((num = fputs(buffer, stream)) != EOF) {
        /* Note that fputs() does not copy the \0 character                   */

         printf("Total number of characters written to file = %i\n", num);
         fclose(stream);
      }
      else                                                /* fputs failed     */
         perror("fputs failed");
   }
   else
      perror("Error opening myfile.dat");
   return 0;

   /****************************************************************************
      The output should be:

      Total number of characters written to file = 26
   ****************************************************************************/
}
```

Related Information

- _cputs
- fgets
- gets
- puts

----

fputwc - Write Wide Character

fputwc - Write Wide Character

Syntax

```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t wc, FILE *stream);
```

Description

fputwc converts the wide character *wc* to a multibyte character and writes it to the output stream pointed to by *stream* at the current position. It also advances the file position indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the stream.

The behavior of fputwc is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent operations on the same stream, undefined results can occur.

After calling fputwc, flush the buffer or reposition the stream pointer before calling a read function for the stream. After reading from the stream, flush the buffer or reposition the stream pointer before calling fputwc, unless EOF has been reached.

Returns

fputwc returns the wide character written. If a write error occurs, the error indicator for the stream is set and fputwc returns WEOF. If an encoding error occurs during conversion from wide character to a multibyte character, fputwc sets errno to EILSEQ and returns WEOF.

Example Code

This example opens a file and uses fputwc to write wide characters to the file.

```
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
```

```
int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"A character string.";
   int     i;

   if (NULL == (stream = fopen("fputwc.out", "w"))) {
      printf("Unable to open: \"fputwc.out\".\n");
      exit(1);
   }

   for (i = 0; wcs[i] != L'\0'; i++) {
      errno = 0;
      if (WEOF == fputwc(wcs[i], stream)) {
         printf("Unable to fputwc() the wide character.\n"
                "wcs[%d] = 0x%lx\n", i, wcs[i]);
         if (EILSEQ == errno)
            printf("An invalid wide character was encountered.\n");
         exit(1);
      }
   }
   fclose(stream);
   return 0;

   /****************************************************************************
      The output file fputwc.out should contain :

      A character string.
   ****************************************************************************/
}
```

Related Information

- fgetwc
- fputc
- fputws

----

fputws - Write Wide-Character String

fputws - Write Wide-Character String

Syntax

```
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t *wcs, FILE *stream);
```

Description

fputws converts the wide-character string *wcs* to a multibyte-character string and writes it to *stream* as a multibyte character string. It does not write the terminating null byte.

The behavior of fputws is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent operations on the same stream, undefined results can occur.

After calling fputws, flush the buffer or reposition the stream pointer before calling a read function for the stream. After a read operation, flush the buffer or reposition the stream pointer before calling fputws, unless EOF has been reached.

Returns

fputws returns a non-negative value if successful. If a write error occurs, the error indicator for the stream is set and fputws returns -1. If an encoding error occurs in converting the wide characters to multibyte characters, fputws sets errno to EILSEQ and returns -1.

Example Code

```
#include <stdio.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"This test string should not return -1";

   if (NULL == (stream = fopen("fputws.out", "w"))) {
      printf("Unable to open: \"fputws.out\".\n");
      exit(1);
   }

   errno = 0;
   if (EOF == fputws(wcs, stream)) {
      printf("Unable to complete fputws() function.\n");
      if (EILSEQ == errno)
         printf("An invalid wide character was encountered.\n");
      exit(1);
   }
   fclose(stream);
   return 0;

   /*****************************************************************************
      The output file fputws.out should contain :

      This test string should not return -1
   *****************************************************************************/
}
```

Related Information

- fgetws
- fputs
- fputwc

----

fread - Read Items

fread - Read Items

Syntax

```
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count,
                FILE *stream);
```

Description

`fread` reads up to *count* items of *size* length from the input *stream* and stores them in the given *buffer*. The position in the file increases by the number of bytes read.

Returns

`fread` returns the number of full items successfully read, which can be less than *count* if an error occurs or if the end-of-file is met before reaching *count*. If *size* or *count* is `0`, `fread` returns zero and the contents of the array and the state of the stream remain unchanged.

Use `ferror` and `feof` to distinguish between a read error and an end-of-file.

Example Code

This example attempts to read `NUM_ALPHA` characters from the file myfile.dat. If there are any errors with either `fread` or `fopen`, a message is printed.

```
#include <stdio.h>

#define  NUM_ALPHA     26

int main(void)
{
   FILE *stream;
   int num;                     /* number of characters read from stream      */

   /* Do not forget that the '\0' char occupies one character too!            */

   char buffer[NUM_ALPHA+1];

   if ((stream = fopen("myfile.dat", "r")) != NULL) {
      num = fread(buffer, sizeof(char), NUM_ALPHA, stream);
      if (num) {                                          /* fread success    */
         printf("Number of characters has been read = %i\n", num);
         buffer[num] = '\0';
         printf("buffer = %s\n", buffer);
         fclose(stream);
      }
      else {                                              /* fread failed     */
         if (ferror(stream))                              /* possibility 1    */
            perror("Error reading myfile.dat");
         else
            if (feof(stream))                             /* possibility 2    */

               perror("EOF found");
      }
   }
   else
      perror("Error opening myfile.dat");
   return 0;

   /****************************************************************************
      The output should be:

      Number of characters has been read = 26
      buffer = abcdefghijklmnopqrstuvwxyz
   ****************************************************************************/
}
```

Related Information

- feof
- ferror
- fopen
- fwrite
- read

----

free - Release Storage Blocks

free - Release Storage Blocks

Syntax

```
#include <stdlib.h>  /* also in <malloc.h> */
void free(void *ptr);
```

Description

`free` frees a block of storage. *ptr* points to a block previously reserved with a call to one of the memory allocation functions (such as calloc, _umalloc, or _trealloc). The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of `realloc`) the block of storage. If *ptr* is `NULL`, `free` simply returns.

Note: Attempting to free a block of storage not allocated with a C memory management function or a block that was previously freed can affect the subsequent reserving of storage and lead to undefined results.

Returns

There is no return value.

Example Code

This example uses `calloc` to allocate storage for *x* array elements and then calls `free` to free them.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long *array;                              /* start of the array            */
   long *index;                                           /* index variable   */
   int i;                                                 /* index variable   */
   int num = 100;                       /* number of entries of the array     */

   printf("Allocating memory for %d long integers.\n", num);


   /* allocate num entries                                                    */

   if ((index = array = calloc(num, sizeof(long))) != NULL) {

     /*.......................................................................*/
     /*.......................................................................*/
            /*  do something with the array                                   */
     /*.......................................................................*/

      free(array);                                        /* deallocates array*/
   }
   else {                                                 /* Out of storage   */
      perror("Error: out of storage");
      abort();
   }
   return 0;

   /****************************************************************************
      The output should be:

      Allocating memory for 100 long integers.
   ****************************************************************************/
}
```

Related Information

- "Managing Memory" in the *VisualAge C++ Programming Guide*
- calloc
- malloc
- realloc

----

freopen - Redirect Open Files

freopen - Redirect Open Files

Syntax

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Description

`freopen` closes the file currently associated with *stream* and reassigns *stream* to the file specified by *filename* . The `freopen` function opens the new file associated with *stream* with the given *mode* , which is a character string specifying the type of access requested for the file. You can also use the `freopen` function to redirect the standard stream files stdin, stdout, and stderr to files that you specify.

If *filename* is an empty string, `freopen` closes and reopens the stream to the new open mode, rather than reassigning it to a new file or device. You can use freopen with no file name specified to change the mode of a standard stream from text to binary without redirecting the stream, for example:

```
   fp = freopen("", "rb", stdin);
```

You can use the same method to change the mode from binary back to text.

Portability Note This method is included in the SAA C definition, but not in the ANSI/ISO C standard.

See fopen for a description of the *mode* parameter.

Returns

`freopen` returns a pointer to the newly opened stream. If an error occurs, `freopen` closes the original file and returns a `NULL` pointer value.

Example Code

This example closes the `stream1` data stream and reassigns its stream pointer. Note that `stream1` and `stream2` will have the same value, but they will not necessarily have the same value as `stream`.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *stream,*stream1,*stream2;

   stream = fopen("myfile.dat", "r");
   stream1 = stream;
   if (NULL == (stream2 = freopen("", "w+", stream1)))
      return EXIT_FAILURE;
   fclose(stream2);
   return 0;
}
```

Related Information

- close
- fclose
- _fcloseall
- fopen
- open
- _sopen

----

frexp - Separate Floating-Point Value

frexp - Separate Floating-Point Value

Syntax

```
#include <math.h>
double frexp(double x, int *expptr);
```

Description

`frexp` breaks down the floating-point value *x* into a term *m* for the mantissa and another term *n* for the exponent, such that $x=m*2n$, and the absolute value of *m* is greater than or equal to `0.5` and less than `1.0` or equal to `0`. `frexp` stores the integer exponent *n* at the location to which *expptr* points.

Returns

`frexp` returns the mantissa term *m*. If *x* is `0`, frexp returns `0` for both the mantissa and exponent. The mantissa has the same sign as the argument *x*. The result of the `frexp` function cannot have a range error.

Example Code

This example decomposes the floating-point value of *x*, 16.4, into its mantissa `0.5125`, and its exponent 5. It stores the mantissa in *y* and the exponent in *n*.

```
#include <math.h>

int main(void)
{
   double x,m;
   int n;

   x = 16.4;
   m = frexp(x, &n);
   printf("The mantissa is %lf and the exponent is %d\n", m, n);
   return 0;

   /*****************************************************************************
      The output should be:

      The mantissa is 0.512500 and the exponent is 5
   *****************************************************************************/
}
```

Related Information

- ldexp
- modf

----

fscanf - Read Formatted Data

fscanf - Read Formatted Data

Syntax

```
#include <stdio.h>
int fscanf (FILE *stream, const char *format-string, argument-list);
```

Description

`fscanf` reads data from the current position of the specified *stream* into the locations given by the entries in *argument-list* , if any. Each entry in *argument-list* must be a pointer to a variable with a type that corresponds to a type specifier in *format-string* .

The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the `scanf` function. See scanf for a description of *format-string* .

In extended mode, the `fscanf` function also reads in the strings `"INFINITY"`, `"INF"`, and `"NAN"` (in uppercase or lowercase) and converts them to the corresponding floating-point value. The sign of the value is determined by the format specification. See Infinity and NaN Support for more information on infinity and NaN values.

Returns

`fscanf` returns the number of fields that it successfully converted and assigned. The return value does not include fields that `fscanf` read but did not assign.

The return value is `EOF` if an input failure occurs before any conversion, or the number of input items assigned if successful.

Example Code

This example opens the file myfile.dat for reading and then scans this file for a string, a long integer value, a character, and a floating-point value.

```
#include <stdio.h>

#define  MAX_LEN       80

int main(void)
{
   FILE *stream;
   long l;
   float fp;
   char s[MAX_LEN+1];
   char c;

   stream = fopen("myfile.dat", "r");

    /* Put in various data.                                                  */

   fscanf(stream, "%s", &s[0]);
   fscanf(stream, "%ld", &l);
   fscanf(stream, "%c", &c);
   fscanf(stream, "%f", &fp);
   printf("string = %s\n", s);
   printf("long %ld\n", l);
   printf("char = %c\n", c);
   printf("float = %f\n", fp);
   return 0;

   /****************************************************************************
      If myfile.dat contains:
      abcdefghijklmnopqrstuvwxyz 343.2.

      The output should be:

      string = abcdefghijklmnopqrstuvwxyz
      long double = 343
      char = .
      float = 2.000000
   ****************************************************************************/
}
```

Related Information

- Infinity and NaN Support
- _cscanf
- fprintf
- scanf

- sscanf

----

fseek - Reposition File Position

fseek - Reposition File Position

Syntax

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);
```

Description

`fseek` changes the current file position associated with *stream* to a new location within the file. The next operation on the *stream* takes place at the new location. On a *stream* open for update, the next operation can be either a reading or a writing operation.

The *origin* must be one of the following constants defined in `<stdio.h>`: compact break=fit.

Origin	Definition
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file

For a binary stream, you can also change the position beyond the end of the file. An attempt to position before the beginning of the file causes an error. If successful, `fseek` clears the end-of-file indicator, even when *origin* is `SEEK_END`, and undoes the effect of any preceding `ungetc` function on the same stream.

Note: For streams opened in text mode, `fseek` has limited use because some system translations (such as those between carriage-return-line-feed and new line) can produce unexpected results. The only `fseek` operations that can be relied upon to work on streams opened in text mode are seeking with an offset of zero relative to any of the origin values or seeking from the beginning of the file with an offset value returned from a call to `ftell`. See the chapter "Performing I/O Operations" in the *VisualAge C++ Programming Guide* for more information.

Returns

`fseek` returns `0` if it successfully moves the pointer. A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

Example Code

This example opens a file myfile.dat for reading. After performing input operations (not shown), fseek moves the file pointer to the beginning of the file.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *stream;
   int result;

   stream = fopen("myfile.dat", "r");

   result = fseek(stream, 0L, SEEK_SET);   /* moves the pointer to the
                                              beginning of the file           */

   if (result)                           /* fail to move the pointer to the   */
```

```
      return EXIT_FAILURE;               /* beginning of the file             */
   fclose(stream);
   return 0;
}
```

Related Information

- fgetpos
- fsetpos
- ftell
- rewind
- ungetc

----

fsetpos - Set File Position

fsetpos - Set File Position

Syntax

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

`fsetpos` moves any file position associated with *stream* to a new location within the file according to the value pointed to by *pos* . The value of *pos* was obtained by a previous call to the `fgetpos` library function.

If successful, `fsetpos` clears the end-of-file indicator, and undoes the effect of any previous `ungetc` function on the same stream.

After the `fsetpos` call, the next operation on a stream in update mode may be input or output.

Returns

If `fsetpos` successfully changes the current position of the file, it returns `0`. A nonzero return value indicates an error.

Example Code

This example opens a file `myfile.dat` for reading. After performing input operations (not shown), `fsetpos` moves the file pointer to the beginning of the file and rereads the first byte.

```
#include <stdio.h>

FILE *stream;

int main(void)
{
   int retcode;
   fpos_t pos,pos1,pos2,pos3;
   char ptr[20];           /* existing file 'myfile.dat' has 20 byte records  */

    /* Open file, get position of file pointer, and read first record         */

   stream = fopen("myfile.dat", "rb");
   fgetpos(stream, &pos);
   pos1 = pos;
   if (!fread(ptr, sizeof(ptr), 1, stream))
      perror("fread error");

    /* Perform a number of read operations - the value of 'pos' changes       */
    /* Re-set pointer to start of file and re-read first record               */

   fsetpos(stream, &pos1);
   if (!fread(ptr, sizeof(ptr), 1, stream))
      perror("fread error");
   fclose(stream);
   return 0;
}
```

Related Information

- fgetpos
- fseek
- ftell
- rewind

----

fstat - Information about Open File

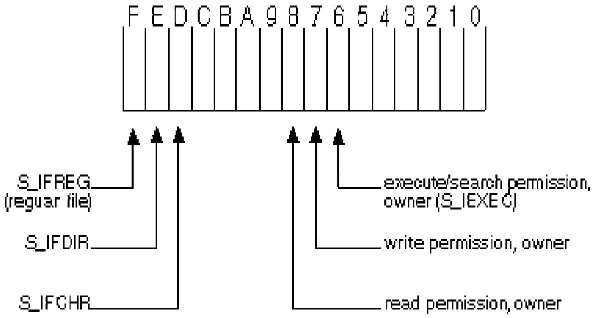
fstat - Information about Open File

Syntax

```
#include <sys\stat.h>
#include <sys\types.h>
int fstat(int handle, struct stat *buffer);
```

Description

fstat obtains information about the open file associated with the given *handle* and stores it in the structure to which *buffer* points. The `<sys\stat.h>` include file defines the `stat` structure. The `stat` structure contains the following fields:

Field	**Value**
st_mode	Bit mask for file mode information. fstat sets the S_IFCHR bit if *handle* refers to a device. The S_IFDIR bit is set if *pathname* specifies a directory. It sets the S_IFREG bit if *handle* refers to an ordinary file. It sets user read/write bits according to the permission mode of the file. The other bits have undefined values.  F E D C B A 9 8 7 6 5 4 3 2 1 0 S_IFREG (reguar file) S_IFDIR S_IFCHR execute/search pemission, owner (S_IEXEC) write permission, owner read pemission, owner
st_dev	Drive number of the disk containing the file.
st_rdev	Drive number of the disk containing the file (same as **st_dev**).
st_nlink	Always 1.
st_size	Size of the file in bytes.
st_atime	Time of last access of file.
st_mtime	Time of last modification of file.

st_ctime   Time of file creation.

There are three additional fields in the `stat` structure for portability to other operating systems; they have no meaning under the OS/2 operating system.

Note: If the given *handle* refers to a device, the size and time fields in the `stat` structure are not meaningful.

Returns

If it obtains the file status information, fstat returns `0`. A return value of -1 indicates an error, and fstat sets errno to EBADF, showing an incorrect file handle.

Example Code

This example uses fstat to report the size of a file named `data`.

```
#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>

int main(void)
{
   struct stat buf;
   FILE *fp;
   int fh,result;
   char *buffer = "A line to output";

   fp = fopen("data", "w+");
   fprintf(fp, "%s", buffer);
   fflush(fp);
   fclose(fp);
   fp = fopen("data", "r");
   if (0 == fstat(fileno(fp), &buf)) {
      printf("file size is %ld\n", buf.st_size);
      printf("time modified is %s\n", ctime(&buf.st_atime));
   }
   else
      printf("Bad file handle\n");
   fclose(fp);
   return 0;

   /****************************************************************************
      The output should be similar to:

      file size is 16
      time modified is Thu May 16 16:08:14 1995
   ****************************************************************************/
}
```

Related Information

- stat

----

ftell - Get Current Position

ftell - Get Current Position

Syntax

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

`ftell` finds the current position of the file associated with *stream* . For a fixed-length binary file, the value returned by `ftell` is an offset relative to the beginning of the *stream* .

Note: For buffered text streams, ftell returns an incorrect file position if the file contains new-line characters instead of carriage-return line-feed combinations. Your file would only contain new-line characters if you previously used it as a binary stream. To avoid this problem, either continue to process the file as a binary stream, or use unbuffered I/O operations.

Returns

ftell returns the current file position. On error, `ftell` returns -1L and sets `errno` to a nonzero value.

Example Code

This example opens the file `myfile.dat` for reading. It reads enough characters to fill half of the buffer and prints out the position in the stream and the buffer.

```
#include <stdio.h>

#define  NUM_ALPHA     26
#define  NUM_CHAR      6

int main(void)
{
   FILE *stream;
   int i;
   char ch;
   char buffer[NUM_ALPHA];
   long position;

   if ((stream = fopen("myfile.dat", "r")) != NULL) {

     /* read into buffer                                                      */

      for (i = 0; (i < NUM_ALPHA/2) && ((buffer[i] = fgetc(stream)) != EOF);
           ++i)
         if (i == NUM_CHAR-1) { /* We want to be able to position the
                                   file pointer to the character in
                                   position NUM_CHAR                          */

            position = ftell(stream);
            printf("Current position of the file is stored.\n");
         }
      buffer[i] = '\0';
      fseek(stream, position, SEEK_SET);
      ch = fgetc(stream);       /* get the character at position NUM_CHAR     */
      fclose(stream);
   }
   else
      perror("Error opening myfile.dat");
   return 0;
}
```

Related Information

- fseek
- fgetpos
- fopen
- fsetpos

_ftime - Store Current Time

_ftime - Store Current Time

Syntax

```
#include <sys\timeb.h>
#include <sys\types.h>
void _ftime(struct timeb *timeptr);
```

Description

_ftime gets the current time and stores it in the structure to which *timeptr* points. The `<sys\timeb.h>` include file contains the definition of the `timeb` structure. It contains four fields:

time — The time in seconds since `00:00:00` Coordinated Universal Time, January 1, 1970.

millitm — The fraction of a second, in milliseconds.

timezone — The difference in minutes between Coordinated Universal Time and local time, going from east to west. _ftime sets the value of `timezone` from the value of the global variable `_timezone`.

dstflag — Nonzero if daylight saving time is currently in effect for the local time zone. For an explanation of how daylight saving time is determined, see tzset.

Returns

There is no return value.

Example Code

This example polls the system clock, converts the current time to a character string, prints the string, and saves the time data in the structure `timebuffer`.

```
#include <sys\types.h>
#include <sys\timeb.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
   struct timeb timebuffer;

   _ftime(&timebuffer);
   printf("the time is %s\n", ctime(&(timebuffer.time)));
   return 0;

   /****************************************************************************
      The output should be similar to:

      the time is Thu May 16 16:08:17 1995
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- localtime
- mktime
- time
- tzset

_fullpath - Get Full Path Name of Partial Path

_fullpath - Get Full Path Name of Partial Path

Syntax

```
#include <stdlib.h>
char *_fullpath(char *pathbuf, char *partialpath, size_t n);
```

Description

_fullpath gets the full path name of the given partial path name *partialpath* , and stores it in *pathbuf* . The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name, including the terminating null character, exceeds *n* characters.

If *pathbuf* is `NULL`, _fullpath uses `malloc` to allocate a buffer of size _MAX_PATH bytes to store the path name.

Returns

_fullpath returns a pointer to *pathbuf* . If an error occurs, _fullpath returns `NULL` and sets errno to one of the following values: compact break=fit.

Value	**Meaning**
ENOMEM	Unable to allocate storage for the buffer.
ERANGE	The path name is longer than *n* characters.
EOS2ERR	A system error occurred. Check _doserrno for the specific OS/2 error code.

Example Code

This example uses _fullpath to get and store the full path name of the current directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(void)
{
   char *retBuffer;

   retBuffer = _fullpath(NULL, ".", 0);
   if (NULL == retBuffer) {
      printf("An error has occurred, errno is set to %d.\n", errno);
   }
   else
      printf("Full path name of current directory is %s.\n", retBuffer);
   return 0;

   /*****************************************************************************
      The output should be similar to:

        Full path name of current directory is D:\BIN.
   *****************************************************************************/
}
```

Related Information

- _getcwd
- _getdcwd
- _makepath
- malloc
- _splitpath

----

fwrite - Write Items

fwrite - Write Items

Syntax

```
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count,
                 FILE *stream);
```

Description

`fwrite` writes up to *count* items, each of *size* bytes in length, from *buffer* to the output *stream* .

Returns

`fwrite` returns the number of full items successfully written, which can be fewer than *count* if an error occurs.

Example Code

This example writes `NUM long` integers to a stream in binary format.

```
#include <stdio.h>

#define  NUM           100

int main(void)
{
   FILE *stream;
   long list[NUM];
   int numwritten;

   stream = fopen("myfile.dat", "w+b");

              /* assign values to list[]                                      */

   numwritten = fwrite(list, sizeof(long), NUM, stream);
   printf("Number of items successfully written : %d\n", numwritten);
   return 0;

   /*****************************************************************************
      The output should be:

      Number of items successfully written : 100
   ****************************************************************************/
}
```

Related Information

- fopen
- fread
- read
- write

----

gamma - Gamma Function

gamma - Gamma Function

Syntax

```
#include <math.h> /* SAA extension to ANSI */
double gamma(double x);
```

Description

gamma computes the natural logarithm of the absolute value of G(x) (ln(|G(x)|)), where

$$G(x) = \int_0^\infty e^{-t} x\, t^{x-1} dt$$

Portability Note According to the SAA standard, *x* must be a positive real value. Under *The Developer's Toolkit* compiler, *x* can also be a negative integer.

Returns

gamma returns the value of ln(|G(x)|). If *x* is a negative value, errno is set to EDOM. If the result causes an overflow, gamma returns HUGE_VAL and sets errno to ERANGE.

Example Code

This example uses gamma to calculate ln(|G(x)|), where x = 42.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x = 42,g_at_x;

   g_at_x = exp(gamma(x));                       /* g_at_x = 3.345253e+49     */
   printf("The value of G(%4.2f) is %7.2e\n", x, g_at_x);
   return 0;

   /****************************************************************************
      The output should be:

      The value of G(42.00) is 3.35e+49
   ****************************************************************************/
}
```

Related Information

- bessel
- erf - erfc

_gcvt - Convert Floating-Point to String

_gcvt - Convert Floating-Point to String

Syntax

```
#include <stdlib.h>
char *_gcvt(double value, int ndec, char *buffer);
```

Description

_gcvt converts a floating-point *value* to a character string pointed to by *buffer* . The *buffer* should be large enough to hold the converted value and a null character (`\0`) that _gcvt automatically adds to the end of the string. There is no provision for overflow.

_gcvt tries to produce *ndec* significant digits in FORTRAN F format. Failing that, it produces *ndec* significant digits in FORTRAN E format. Trailing zeros might be suppressed in the conversion if they are significant.

A FORTRAN F number has the following format:

```
>>             digit    .                  ><
       +                      digit
```

A FORTRAN E number has the following format:

```
>>          digit  .    digit    E        digit                 ><
       +                              +              digit
```

_gcvt also converts NaN and infinity values to the strings `NAN` and `INFINITY`, respectively. For more information on NaN and infinity values, see Infinity and NaN Support.

Warning: For each thread, _ecvt, _fcvt and _gcvt use a single, dynamically allocated buffer for the conversion. Any subsequent call that the same thread makes to these functions destroys the result of the previous call.

Returns

_gcvt returns a pointer to the string of digits. If it cannot allocate memory to perform the conversion, _gcvt returns an empty string and sets errno to ENOMEM.

Example Code

This example converts the value -3.1415e3 to a character string and places it in the character array `buffer1`. It then converts the macro value `_INF` to a character string and places it in `buffer2`.

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>                  /* for the definition of _INF             */

int main(void)
{
   char buffer1[10],buffer2[10];

   _gcvt(-3.1415e3, 7, buffer1);
   printf("The first result is %s \n", buffer1);
   _gcvt(_INF, 5, buffer2);
   printf("The second result is %s \n", buffer2);
   return 0;

   /****************************************************************************
      The output should be:

      The first result is -3141.5
      The second result is INFINITY
   ****************************************************************************/
}
```

Related Information

- _ecvt
- _fcvt
- Infinity and NaN Support

----

getc - getchar - Read a Byte

getc - getchar - Read a Byte

Syntax

```
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

Description

`getc` reads a single byte from the current *stream* position and advances the *stream* position to the next byte. `getchar` is identical to `getc(stdin)`.

`getc` is equivalent to fgetc except that, if it is implemented as a macro, `getc` can evaluate *stream* more than once. Therfore, the *stream* argument to `getc` should not be an expression with side effects.

Returns

`getc` and `getchar` return the value read. A return value of `EOF` indicates an error or end-of-file condition. Use `ferror` or `feof` to determine whether an error or an end-of-file condition occurred.

Example Code

This example gets a line of input from the stdin stream. You can also use `getc(stdin)` instead of `getchar()` in the `for` statement to get a line of input from stdin.

```
#include <stdio.h>

#define  LINE          80

int main(void)
{
   char buffer[LINE+1];
   int i;
   int ch;

   printf("Please enter string\n");

   /* Keep reading until either:
      1. the length of LINE is exceeded  or
      2. the input character is EOF  or
      3. the input character is a newline character
                                                                              */

   for (i = 0; (i < LINE) && ((ch = getchar()) != EOF) && (ch != '\n'); ++i)
      buffer[i] = ch;
   buffer[i] = '\0';         /* a string should always end with '\0' !        */
   printf("The string is %s\n", buffer);
   return 0;
   /**************************************************************************
      The output should be similar to:

      Please enter string
      hello world
      The string is hello world
   **************************************************************************/
}
```

Related Information

- fgetc
- _getch - _getche
- putc - putchar
- ungetc
- gets

_getch - _getche - Read Character from Keyboard

_getch - _getche - Read Character from Keyboard

Syntax

```
#include <conio.h>
int _getch(void);
int _getche(void);
```

Description

_getch reads a single character from the keyboard, without echoing. _getche reads a single character from the keyboard and displays the character read. Neither function can be used to read Ctrl-Break.

You can use _kbhit to test if a keystroke is waiting in the buffer. If you call _getch or _getche without first calling _kbhit, the program waits for a key to be pressed.

Returns

_getch and _getche return the character read. To read a function key or cursor-moving key, you must call _getch and _getche twice; the first call returns `0` or `E0H`, and the second call returns the particular extended key code.

Example Code

This example gets characters from the keyboard until it finds the character 'x'.

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
   int ch;

   printf("Type in some letters.\n");
   printf("If you type in an 'x', the program ends.\n");
   for (; ; ) {
      ch = _getch();
      if ('x' == ch) {
         _ungetch(ch);
         break;
      }
      _putch(ch);
   }
   ch = _getch();
```

```
      printf("\nThe last character was '%c'.", ch);
      return 0;

      /****************************************************************************
         Here is the output from a sample run:

         Type in some letters.
         If you type in an 'x', the program ends.
         One Two Three Four Five Si
         The last character was 'x'.
      ****************************************************************************/
   }
```

Related Information

- _cgets
- fgetc
- getc - getchar
- _kbhit
- _putch
- _ungetch

_getcwd - Get Path Name of Current Directory

_getcwd - Get Path Name of Current Directory

Syntax

```
#include <direct.h>
char *_getcwd(char *pathbuf, int n);
```

Description

_getcwd gets the full path name of the current working directory and stores it in the buffer pointed to by *pathbuf*. The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name, including the terminating null character, exceeds *n* characters.

If the *pathbuf* argument is NULL, _getcwd uses malloc to reserve a buffer of at least *n* bytes to store the path name. If the current working directory string is more than *n* bytes, a large enough buffer will be allocated to contain the string. You can later free this buffer using the _getcwd return value as the argument to free.

Returns

_getcwd returns *pathbuf*. If an error occurs, _getcwd returns NULL and sets errno to one of the following values: compact break=fit.

Value	Meaning
ENOMEM	Not enough storage space available to reserve *n* bytes (when *pathbuf* is NULL).
ERANGE	The path name is longer than *n* characters.
EOS2ERR	An OS/2 call failed. Use _doserrno to obtain more information about the return code.

Example Code

This example stores the name of the current working directory (up to _MAX_PATH characters) in a buffer. The value of _MAX_PATH is defined in <stdlib.h>.

```
#include <direct.h>
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char buffer[_MAX_PATH];

   if (NULL == getcwd(buffer, _MAX_PATH))
      perror("getcwd error");
   printf("The current directory is %s.\n", buffer);
   return 0;

   /****************************************************************************
      The output should be similar to:

      The current directory is E:\C_OS2\MIG_XMPS
   ****************************************************************************/
}
```

Related Information

- chdir
- _fullpath
- _getdcwd
- _getdrive
- malloc

_getdcwd - Get Full Path Name of Current Directory

_getdcwd - Get Full Path Name of Current Directory

Syntax

```
#include <direct.h>
char *_getdcwd(int drive, char *pathbuf, int n);
```

Description

_getdcwd gets the full path name for the current directory of the specified *drive* , and stores it in the location pointed to by *pathbuf* . The *drive* argument is an integer value representing the drive (`A:` is 1, `B:` is 2, and so on).

The integer argument *n* specifies the maximum length for the path name. An error occurs if the length of the path name, including the terminating null character, exceeds *n* characters.

If the *pathbuf* argument is `NULL`, _getdcwd uses `malloc` to reserve a buffer of at least *n* bytes to store the path name. If the current working directory string is more than *n* bytes, a large enough buffer will be allocated to contain the string. You can later free this buffer using the _getdcwd return value as the argument to `free`.

Alternatives to this function are the DosQueryCurrentDir and DosQueryCurrentDisk functions.

Returns

_getdcwd returns *pathbuf* . If an error occurs, _getdcwd returns `NULL` and sets `errno` to one of the following values: compact break=fit.

Value	Meaning
ENOMEM	Not enough storage space available to reserve *n* bytes (when *pathbuf* is `NULL`).
ERANGE	The path name is longer than *n* characters.
EOS2ERR	An OS/2 call failed. Use _doserrno to obtain more information about the return code.

Example Code

This example uses _getdcwd to obtain the current working directory of drive C.

```
#include <stdio.h>
#include <direct.h>
#include <errno.h>

int main(void)
{
   char *retBuffer;

   retBuffer = _getdcwd(3, NULL, 0);
   if (NULL == retBuffer)
      printf("An error has occurred, errno is set to %d.\n", errno);
   else
      printf("%s is the current working directory in drive C.\n", retBuffer);
   return 0;

   /****************************************************************************
      The output should be similar to:

      C:\ is the current working directory in drive C.
   ****************************************************************************/
}
```

Related Information

- chdir
- _chdrive
- _fullpath
- _getcwd
- _getdrive
- malloc

_getdrive - Get Current Working Drive

_getdrive - Get Current Working Drive

Syntax

```
#include <direct.h>
int _getdrive(void);
```

Description

_getdrive gets the drive number for the current working drive.

An alternative to this function is the DosQueryCurrentDisk call.

Returns

_getdrive returns an integer corresponding to alphabetical position of the letter representing the current working drive. For example, `A:` is 1, `B:` is 2, `J:` is 10, and so on.

Example Code

This example gets and prints the current working drive number.

```
#include <stdio.h>
```

```
#include <direct.h>

int main(void)
{
   printf("Current working drive is %d.\n", _getdrive());
   return 0;

   /******************************************************************************
      The output should be similar to:

      Current working drive is 5.
   ******************************************************************************/
}
```

Related Information

- chdir
- _chdrive
- _getcwd
- _getdcwd

----

getenv - Search for Environment Variables

getenv - Search for Environment Variables

Syntax

```
#include <stdlib.h>
char *getenv(const char *varname);
```

Description

`getenv` searches the list of environment variables for an entry corresponding to *varname* .

Returns

`getenv` returns a pointer to the environment table entry containing the current string value of *varname* . The return value is `NULL` if the given variable is not currently defined or if the system does not support environment variables.

You should copy the string that is returned because it may be written over by a subsequent call to `getenv`.

Example Code

In this example, *pathvar* points to the value of the `PATH` environment variable.

```
#include <stdlib.h>

int main(void)
{
   char *pathvar;

   pathvar = getenv("PATH");
   if (NULL == pathvar)
      printf("Environment variable PATH is not defined.\n");
   else
      printf("Path set by environment variable PATH is successfully stored.\n");
   return 0;

   /******************************************************************************
      The output should be:

      Path set by environment variable PATH is successfully stored.
   ****************************************************************************/
}
```

Related Information

- putenv

----

getpid - Get Process Identifier

getpid - Get Process Identifier

Syntax

```
#include <process.h>
int getpid(void);
```

Description

getpid gets the process identifier that uniquely identifies the calling process.

Note: In earlier releases of C Set ++, getpid began with an underscore (`_getpid`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_getpid` to getpid for you.

Returns

getpid function the process identifier as an integer value. There is no error return value.

Example Code

This example prints the process identifier:

```
#include <process.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   printf("Process identifier is %05d\n", getpid());
   return 0;

   /****************************************************************************
      The output should be similar to:

      Process identifier is 00242
   ****************************************************************************/
}
```

Related Information

- _cwait
- execl - _execvpe
- _spawnl - _spawnvpe
- wait

----

gets - Read a Line

gets - Read a Line

Syntax

```
#include <stdio.h>
char *gets(char *buffer);
```

Description

`gets` reads a line from the standard input stream stdin and stores it in *buffer*. The line consists of all characters up to and including the first new-line character (`\n`) or `EOF`. `gets` then replaces the new-line character, if read, with a null character (`\0`) before returning the line.

Returns

If successful, `gets` returns its argument. A `NULL` pointer return value indicates an error or an end-of-file condition with no characters read. Use `ferror` or `feof` to determine which of these conditions occurred. If there is an error, the value stored in *buffer* is undefined. If an end-of-file condition occurs, *buffer* is not changed.

Example Code

This example gets a line of input from stdin.

```
#include <stdio.h>

#define  MAX_LINE      100

int main(void)
{
   char line[MAX_LINE];
   char *result;

   if ((result = gets(line)) != NULL) {
      if (ferror(stdin))
         perror("Error");
      printf("Input line : %s\n", result);
   }
   return 0;

   /****************************************************************************
      For the following input:
      This is a test for function gets.

      The output should be:
      Input line : This is a test for function gets.
   ****************************************************************************/
}
```

Related Information

- _cgets
- fgets
- feof
- ferror
- fputs
- getc - getchar
- puts

----

getwc - Read Wide Character from Stream

getwc - Read Wide Character from Stream

Syntax

```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

getwc reads the next multibyte character from *stream* , converts it to a wide character, and advances the associated file position indicator for *stream* .

getwc is equivalent to fgetwc except that, if it is implemented as a macro, it can evaluate *stream* more than once. Therefore, the argument should never be an expression with side effects.

The behavior of getwc is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

After calling getwc, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling getwc.

Returns

getwc returns the next wide character from the input stream, or WEOF. If an error occurs, getwc sets the error indicator. If getwc encounters the end-of-file, it sets the EOF indicator. If an encoding error occurs during conversion of the multibyte character, getwc sets errno to EILSEQ.

Use ferror or feof to determine whether an error or an EOF condition occurred. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

Example Code

This example opens a file and uses getwc to read wide characters from the file.

```
#include <errno.h>
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wint_t  wc;

   if (NULL == (stream = fopen("getwc.dat", "r"))) {
      printf("Unable to open: \"getwc.dat\".\n");
      exit(1);
   }

   errno = 0;
   while (WEOF !=(wc = getwc(stream)))
      printf("wc = %lc\n", wc);

   if (EILSEQ == errno) {
      printf("An invalid wide character was encountered.\n");
      exit(1);
   }
   fclose(stream);
   return 0;

   /***************************************************************************
      Assuming the file getwc.dat contains:
```

```
      Hello world!

      The output should be:

      wc = H
      wc = e
      wc = l
      wc = l
      wc = o
      :
   ****************************************************************************/
}
```

Related Information

- fgetwc
- getwchar
- getc - getchar
- _getch - _getche
- putwc

----

getwchar - Get Wide Character from stdin

getwchar - Get Wide Character from stdin

Syntax

```
#include <wchar.h>
wint_t getwchar(void);
```

Description

getwchar reads the next multibyte character from stdin, converts it to a wide character, and advances the associated file position indicator for stdin. A call to getwchar is equivalent to a call to `getwc(stdin)`.

The behavior of getwchar is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

Returns

getwchar returns the next wide character from stdin or WEOF. If getwchar encounters EOF, it sets the EOF indicator for the stream and returns WEOF. If a read error occurs, the error indicator for the stream is set and getwchar returns WEOF. If an encoding error occurs during the conversion of the multibyte character to a wide character, getwchar sets errno to EILSEQ and returns WEOF.

Use ferror or feof to determine whether an error or an EOF condition occurred. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

Example Code

This example uses getwchar to read wide characters from the keyboard, then prints the wide characters.

```
#include <errno.h>
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wint_t  wc;

   errno = 0;
   while (WEOF != (wc = getwchar()))
      printf("wc = %lc\n", wc);

   if (EILSEQ == errno) {
      printf("An invalid wide character was encountered.\n");
      exit(1);
   }
   return 0;

   /****************************************************************************
      Assuming you enter: abcde^Z                        (note: ^Z is CNTRL-Z)

      The output should be:

      wc = a
      wc = b
      wc = c
      wc = d
      wc = e
   ****************************************************************************/
}
```

Related Information

- fgetc
- getc - getchar
- _getch - _getche
- getwc

----

gmtime - Convert Time

gmtime - Convert Time

Syntax

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

Description

The `gmtime` function breaks down the *time* value and stores it in a `tm` structure, defined in `time.h`. The structure pointed to by the return value reflects Coordinated Universal Time, not local time. The value *time* is usually obtained from a call to `time`.

The fields of the `tm` structure include: compact break=fit.

`tm_sec`	Seconds (0-61)
`tm_min`	Minutes (0-59)
`tm_hour`	Hours (0-23)
`tm_mday`	Day of month (1-31)
`tm_mon`	Month (0-11; January = 0)
`tm_year`	Year (current year minus 1900)
`tm_wday`	Day of week (0-6; Sunday = 0)
`tm_yday`	Day of year (0-365; January 1 = 0)
`tm_isdst`	Zero if Daylight Saving Time is not in effect; positive if Daylight Saving Time is in

effect; negative if the information is not available.

Returns

`gmtime` returns a pointer to the resulting `tm` structure. It returns `NULL` if Coordinated Universal Time is not available.

Note:

- The range (0-61) for `tm_sec` allows for as many as two leap seconds.
- The `gmtime` and `localtime` functions may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
- The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Example Code

This example uses `gmtime` to adjust a `time_t` representation to a Coordinated Universal Time character string, and then converts it to a printable string using `asctime`.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   time_t ltime;

   time(&ltime);
   printf("Coordinated Universal Time is %s\n", asctime(gmtime(&ltime)));
   return 0;

   /****************************************************************************
      The output should be similar to:

      Coordinated Universal Time is Mon Sep 16 21:44 1995
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- localtime
- mktime
- time

_heapchk - Validate Default Memory Heap

_heapchk - Validate Default Memory Heap

Syntax

```
#include <malloc.h>
int _heapchk(void);
```

Description

_heapchk checks the default storage heap for minimal consistency by checking all allocated and freed objects on the heap.

A heap-specific version of this function, _uheapchk, is also available.

Note: Using the _heapchk, _heapset, and _heap_walk functions (and their heap-specific equivalents) may add overhead to each object allocated from the heap.

Returns

_heapchk returns one of the following values, defined in <malloc.h>:

_HEAPBADBEGIN	The heap specified is not valid. (Only occurs if you changed the default heap and then later closed or destroyed it.)
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	The heap appears to be consistent.

Example Code

This example performs some memory allocations, then calls _heapchk to check the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

int main(void)
{
   char *ptr;
   int  rc;

   if (NULL == (ptr = malloc(10))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   *(ptr - 1) = 'x';      /* overwrites storage that was not allocated */

   if (_HEAPOK != (rc = _heapchk())) {
      switch(rc) {
         case _HEAPEMPTY:
            puts("The heap has not been initialized.");
            break;
         case _HEAPBADNODE:
            puts("A memory node is corrupted or the heap is damaged.");
            break;
         case _HEAPBADBEGIN:
            puts("The heap specified is not valid.");
            break;
      }
      exit(rc);
   }
   free(ptr);
   return 0;

   /****************************************************************************
      The output should be similar to :

      A memory node is corrupted or the heap is damaged.
   ****************************************************************************/
}
```

Related Information

- "Managing Memory" in the *VisualAge C++ Programming Guide*
- _uheapchk
- _heapmin
- _heapset
- _heap_walk

_heapmin - Release Unused Memory from Default Heap

_heapmin - Release Unused Memory from Default Heap

Syntax

```
#include <stdlib.h>  /* also in <malloc.h> */
int _heapmin(void);
```

Description

_heapmin returns all unused memory from the default run-time heap to the operating system.

Heap-specific versions of this function (_uheapmin) are also available. _heapmin always operates on the default heap.

Note: If you create your own heap and make it the default heap, _heapmin calls the release function that you provide to return the memory.

Returns

_heapmin returns 0 if successful; if not, it returns -1.

Example Code

This example shows how to use the _heapmin function.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   if (_heapmin())
      printf("_heapmin failed.\n");
   else
      printf("_heapmin was successful.\n");
   return 0;

   /****************************************************************************
      The output should be:

     _heapmin was successful.
   ****************************************************************************/
}
```

Related Information

- "Managing Memory" in the *VisualAge C++ Programming Guide*
- _uheapmin

_heapset - Validate and Set Default Heap

_heapset - Validate and Set Default Heap

Syntax

```
#include <malloc.h>
int _heapset(unsigned int fill);
```

Description

_heapset checks the default storage heap for minimal consistency by checking all allocated and freed objects on the heap (similar to _heapchk). It then sets each byte of the heap's free objects to the value of *fill*.

Using _heapset can help you locate problems where your program continues to use a freed pointer to an object. After you set the free heap to a specific value, when your program tries to interpret the set values in the freed object as data, unexpected results occur, indicating a problem.

A heap-specific version of this function, _uheapset, is also available.

Note: Using the _heapchk, _heapset, and _heap_walk functions (and their heap-specific equivalents) may add overhead to each object allocated from the heap.

Returns

_heapset returns one of the following values, defined in `<malloc.h>`:

_HEAPBADBEGIN	The heap specified is not valid; it may have been closed or destroyed.
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	The heap appears to be consistent.

Example Code

This example allocates and frees memory, then uses _heapset to set the free heap to X. It checks the return code from _heapset to ensure the heap is still valid.

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

int main (void)
{
   char  *ptr;
   int   rc;

   if (NULL == (ptr = malloc(10))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   memset(ptr,'A',5);
   free(ptr);

   if (_HEAPOK != (rc = _heapset('X'))) {
      switch(rc) {
         case _HEAPEMPTY:
            puts("The heap has not been initialized.");
            break;
         case _HEAPBADNODE:
            puts("A memory node is corrupted or the heap is damaged.");
            break;
         case _HEAPBADBEGIN:
            puts("The heap specified is not valid.");
            break;
      }
      exit(rc);
   }
   return 0;
}
```

Related Information

- "Managing Memory" in the *VisualAge C++ Programming Guide*
- _heapchk
- _heapmin
- _heap_walk
- _uheapset

_heap_walk - Return Information about Default Heap

_heap_walk - Return Information about Default Heap

Syntax

```
#include <malloc.h>
int _heap_walk(int (*callback_fn)(const void *object, size_t size,
                                                               int flag, int status,
                                                               const char* file, int line) );
```

Description

_heap_walk traverses the default heap, and for each allocated or freed object, it calls the *callback_fn* function that you provide. For each object, it passes your function:

object A pointer to the object.

size The size of the object.

flag The value _USEDENTRY if the object is currently allocated, or _FREEENTRY if the object has been freed. (Both values are defined in `<malloc.h>`.)

status One of the following values, defined in `<malloc.h>`, depending on the status of the object:

_HEAPBADBEGIN The heap specified is not valid; it may have been closed or destroyed.

_HEAPBADNODE A memory node is corrupted, or the heap is damaged.

_HEAPEMPTY The heap has not been initialized.

_HEAPOK The heap appears to be consistent.

file The name of the file where the object was allocated.

line The line where the object was allocated.

_heap_walk provides information about all objects, regardless of which memory management functions were used to allocate them. However, the *file* and *line* information are only available if the object was allocated and freed using the debug versions of the memory management functions. Otherwise, *file* is NULL and *line* is `0`.

_heap_walk calls *callback_fn* for each object until one of the following occurs:

- All objects have been traversed.
- *callback_fn* returns a nonzero value to _heap_walk.
- It cannot continue because of a problem with the heap.

You may want to code your *callback_fn* to return a nonzero value if the status of the object is not _HEAPOK. Even if *callback_fn* returns `0` for an object that is corrupted, _heap_walk cannot continue because of the state of the heap and returns to its caller.

You can use *callback_fn* to process the information from _heap_walk in various ways. For example, you may want to print the information to a file or use it to generate your own error messages. You can use the information to look for memory leaks and objects incorrectly allocated or freed from the heap. It can be especially useful to call _heap_walk when _heapchk returns an error.

A heap-specific version of this function, _uheap_walk, is also available.

Note:

- Using the _heapchk, _heapset, and _heap_walk functions (and their heap-specific equivalents) may add overhead to each object allocated from the heap.

- _heap_walk locks the heap while it traverses it, to ensure that no other operations use the heap until _heap_walk finishes. As a result, in your *callback_fn* , you cannot call any critical functions in the run-time library, either explicitly or by calling another function that calls a critical function. See the *VisualAge C++ Programming Guide* for a list of critical functions.

Returns

_heap_walk returns the last value of *status* to the caller.

Example Code

This example allocates some memory, then uses _heap_walk to walk the heap and pass information about allocated objects to the callback function `callback_function`. `callback_function` then checks the information and keeps track of the number of allocated objects.

```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>

int callback_function(const void *pentry, size_t sz, int useflag, int status,
                      const char *filename, size_t line)
{
   if (_HEAPOK != status) {
      puts("status is not _HEAPOK.");
      exit(status);
   }
   if (_USEDENTRY == useflag)
      printf("allocated  %p     %u\n", pentry, sz);
   else
      printf("freed      %p     %u\n", pentry, sz);
   return 0;
}

int main(void)
{
   char  *p1, *p2, *p3;

   if (NULL == (p1 = malloc(100)) ||
       NULL == (p2 = malloc(200)) ||
       NULL == (p3 = malloc(300))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   free(p2);
   puts("usage      address   size\n-----      -------   ----");

   _heap_walk(callback_function);

   free(p1);
   free(p3);
   return 0;

   /****************************************************************************
      The output should be similar to :

      usage      address   size
      -----      -------   ----
      allocated  73A10     300
      allocated  738B0     100
       :
       :
      freed      73920     224
   ****************************************************************************/
}
```

Related Information

- "Managing Memory" in the *VisualAge C++ Programming Guide*
- "Debugging Your Heaps" in the *VisualAge C++ Programming Guide*
- _heapchk
- _heapmin
- _heapset
- _uheap_walk

----

hypot - Calculate Hypotenuse

hypot - Calculate Hypotenuse

Syntax

```
#include <math.h>
double hypot(double side1, double side2);
```

Description

hypot calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides *side1* and *side2*. A call to hypot is equivalent to:

```
   sqrt(side1 * side1 + side2 * side2);
```

Returns

hypot returns the length of the hypotenuse. If an overflow results, hypot sets errno to ERANGE and returns the value HUGE_VAL. If an underflow results, hypot sets errno to ERANGE and returns zero.

Example Code

This example calculates the hypotenuse of a right-angled triangle with sides of 3.0 and 4.0.

```
#include <math.h>

int main(void)
{
   double x,y,z;

   x = 3.0;
   y = 4.0;
   z = hypot(x, y);
   printf("The hypotenuse of the triangle with sides %lf and %lf"
   " is %lf\n", x, y, z);
   return 0;

   /****************************************************************************
      The output should be:

     The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000
   ****************************************************************************/
}
```

Related Information

- sqrt

----

iconv - Convert Characters to New Code Set

iconv - Convert Characters to New Code Set

Syntax

```
#include <iconv.h>
size_t iconv(iconv_t cd, char **inbuf, size_t *insize,
             char **outbuf, size_t *outsize);
```

Description

iconv converts a sequence of characters in one encoded character set, in the array indirectly pointed to by *inbuf* and converts them to a sequence of corresponding characters in another encoded character set. It stores the corresponding sequence in the array indirectly pointed to by *outbuf*. *cd* is the conversion descriptor returned from iconv_open that describes which codesets are used in the conversion.

inbuf points to a variable that points to the first character of input, and *insize* indicates the number of bytes of input. *outbuf* points to a variable that points to the first character of output, and *outsize* indicates the number of available bytes for output.

As iconv converts the characters, it updates the variable pointed to by *inbuf* to point to the next byte in the input buffer, and updates the variable pointed to by *outbuf* to the byte following the last successfully converted character. It also decrements *insize* and *outsize* to reflect the number of bytes of input remaining and the number of bytes still available for output, respectively. If the entire input string is converted, *insize* will be `0`; if an error stops the conversion, *insize* will have a nonzero value.

Sharing a conversion descriptor in iconv across multiple threads may result in undefined behavior.

Returns

iconv returns the number of nonidentical conversions (substitutions). If an error occurs, conversion stops after the previous successfully converted character; iconv returns `(size_t)-1`, and sets errno to one of the following values: compact break=fit.

Value	**Meaning**
EILSEQ	A sequence of input bytes does not form a valid character in the specified encoded character set.
E2BIG	*outbuf* is not large enough to hold the converted value.
EINVAL	The input ends with an incomplete character.
EBADF	*cd* is not a valid conversion descriptor.

Example Code

This example converts an array of characters coded in encoded character set IBM-850 (in `in`) to an array of characters coded in encoded character set IBM-037 (stored in `out`).

```
#include <iconv.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   const char   fromcode[] = "IBM-850";
   const char   tocode[] = "IBM-037";
   iconv_t      cd;
   char         in[] = "ABCDEabcde";
   size_t       in_size;
   char         *inptr = in;
   char         out[100];
   size_t       out_size = sizeof(out);
   char         *outptr = out;
   int          i;

   if ((iconv_t)(-1) == (cd = iconv_open(tocode, fromcode))) {
      printf("Failed to iconv_open %s to %s.\n", fromcode, tocode);
      exit(EXIT_FAILURE);
   }
   /* Convert the first 3 characters in array "in". */
   in_size = 3;
   if ((size_t)(-1) == iconv(cd, &inptr, &in_size, &outptr, &out_size)) {
      printf("Fail to convert first 3 characters to new code set.\n");
      exit(EXIT_FAILURE);
   }
   /* Convert the rest of the characters in array "in". */
   in_size = sizeof(in) - 3;
   if ((size_t)(-1) == iconv(cd, &inptr, &in_size, &outptr, &out_size)) {
      printf("Fail to convert the rest of the characters to new code set.\n");
      exit(EXIT_FAILURE);
   }
   *outptr = '\0';
   printf("The hex representation of string %s are:\n  In codepage %s ==> ",
          in, fromcode);
   for (i = 0; in[i] != '\0'; i++) {
      printf("0x%02x ", in[i]);
   }
   printf("\n  In codepage %s ==> ", tocode);
   for (i = 0; out[i] != '\0'; i++) {
      printf("0x%02x ", out[i]);
   }
   if (-1 == iconv_close(cd)) {
      printf("Fail to iconv_close.\n");
      exit(EXIT_FAILURE);
   }
   return 0;

   /*****************************************************************************
      The output should be similar to :

      The hex representation of string ABCDEabcde are:
        In codepage IBM-850 ==> 0x41 0x42 0x43 0x44 0x45 0x61 0x62 0x63 0x64 0x65
        In codepage IBM-037 ==> 0xc1 0xc2 0xc3 0xc4 0xc5 0x81 0x82 0x83 0x84 0x85
   *****************************************************************************/
}
```

Related Information

- iconv_close
- iconv_open
- setlocale
- "Internationalization Services" in the *VisualAge C++ Programming Guide*

----

iconv_close - Remove Conversion Descriptor

iconv_close - Remove Conversion Descriptor

Syntax

```
#include <iconv.h>
int iconv_close(iconv_t cd);
```

Description

iconv_close deallocates the conversion descriptor *cd* and all other associated resources allocated by the iconv_open function.

Returns

If successful, iconv_close returns `0`. Otherwise, iconv_close returns -1 is returned and sets errno to indicate the cause of the error. If *cd* is not a valid descriptor, an error occurs and iconv_close sets errno to EBADF.

Example Code

This example shows how you would use iconv_close to remove a conversion descriptor.

```
#include <iconv.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   const char   fromcode[] = "IBM-850";
   const char   tocode[] = "IBM-863";
   iconv_t      cd;

   if ((iconv_t)(-1) == (cd = iconv_open(tocode, fromcode))) {
      printf("Failed to iconv_open %s to %s.\n", fromcode, tocode);
      exit(EXIT_FAILURE);
   }
   if (-1 == iconv_close(cd)) {
      printf("Fail to iconv_close.\n");
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- iconv
- iconv_open
- setlocale
- "Internationalization Services" in the *VisualAge C++ Programming Guide*

----

iconv_open - Create Conversion Descriptor

iconv_open - Create Conversion Descriptor

Syntax

```
#include <iconv.h>
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

Description

iconv_open performs all the initialization needed to convert characters from the encoded character set specified in the array pointed to by *fromcode* to the encoded character set specified in the array pointed to by *tocode* . It creates a conversion descriptor that relates the two encoded character sets. You can then use the conversion descriptor with the iconv function to convert characters between the codesets.

The conversion descriptor remains valid until you close it with iconv_close.

For information about the settings of *fromcode* , *tocode* , and their permitted combinations, see the section on internationalization in the *VisualAge C++ Programming Guide* .

Returns

If successful, iconv_open returns a conversion descriptor of type `iconv_t`. Otherwise, it returns `(iconv_t)-1`, and sets errno to indicate the error. If you cannot convert between the encoded character sets specified, an error occurs and iconv_open sets errno to EINVAL.

Example Code

This example shows how to use iconv_open, iconv, and iconv_close to convert characters from one codeset to another.

```
#include <iconv.h>
#include <stdlib.h>
```

```
#include <stdio.h>

int main(void)
{
   const char   fromcode[] = "IBM-932";
   const char   tocode[] = "IBM-930";
   iconv_t      cd;
   char         in[] = "\x81\x40\x81\x80";
   size_t       in_size = sizeof(in);
   char         *inptr = in;
   char         out[100];
   size_t       out_size = sizeof(out);
   char         *outptr = out;
   int          i;

   if ((iconv_t)(-1) == (cd = iconv_open(tocode, fromcode))) {
      printf("Failed to iconv_open %s to %s.\n", fromcode, tocode);
      exit(EXIT_FAILURE);
   }
   if ((size_t)(-1) == iconv(cd, &inptr, &in_size, &outptr, &out_size)) {
      printf("Fail to convert characters to new code set.\n");
      exit(EXIT_FAILURE);
   }
   *outptr = '\0';
   printf("The hex representation of string %s are:\n  In codepage %s ==> ",
          in, fromcode);
   for (i = 0; in[i] != '\0'; i++) {
      printf("0x%02x ", in[i]);
   }
   printf("\n  In codepage %s ==> ", tocode);
   for (i = 0; out[i] != '\0'; i++) {
      printf("0x%02x ", out[i]);
   }
   if (-1 == iconv_close(cd)) {
      printf("Fail to iconv_close.\n");
      exit(EXIT_FAILURE);
   }
   return 0;

   /****************************************************************************
      The output should be similar to :

      The hex representation of string ×@×  are:
        In codepage IBM-932 ==> 0x81 0x40 0x81 0x80
        In codepage IBM-930 ==> 0x0e 0x40 0x40 0x44 0x7b 0x0f
   ****************************************************************************/
}
```

Related Information

- iconv
- iconv_close
- setlocale
- "Internationalization Services" in the *VisualAge C++ Programming Guide*

----

isalnum to isxdigit - Test Integer Value

isalnum to isxdigit - Test Integer Value

Syntax

```
#include <ctype.h>
                              /* test for:                            */
int isalnum(int c);  /* alphanumeric character               */
int isalpha(int c);  /* alphabetic character                 */
int iscntrl(int c);  /* control character                    */
int isdigit(int c);  /* decimal digit                        */
int isgraph(int c);  /* printable character, excluding space */
int islower(int c);  /* lowercase character                  */
int isprint(int c);  /* printable character, including space */
int ispunct(int c);  /* nonalphanumeric printable character, excluding space */
int isspace(int c);  /* whitespace character                 */
int isupper(int c);  /* uppercase character                  */
int isxdigit(int c); /* hexadecimal digit                    */
```

Description

These functions test a given integer value *c* to determine if it has a certain property as defined by the LC_CTYPE category of your current locale. The value of *c* must be representable as an **unsigned char**, or EOF.

The functions test for the following:

isalnum
Alphanumeric character (uppercase or lowercase letter, or decimal digit), as defined in the locale source file in the `alnum` class of the LC_CTYPE category of the current locale.

isalpha
Alphabetic character, as defined in the locale source file in the `alpha` class of the LC_CTYPE category of the current locale.

iscntrl
Control character, as defined in the locale source file in the `cntrl` class of the LC_CTYPE category of the current locale.

isdigit
Decimal digit (`0` through 9), as defined in the locale source file in the `digit` class of the LC_CTYPE category of the current locale.

isgraph
Printable character, excluding the space character, as defined in the locale source file in the `graph` class of the LC_CTYPE category of the current locale.

islower
Lowercase letter, as defined in the locale source file in the `lower` class of the LC_CTYPE category of the current locale.

isprint
Printable character, including the space character, as defined in the locale source file in the `print` class of the LC_CTYPE category of the current locale.

ispunct
Nonalphanumeric printable character, excluding the space character, as defined in the locale source file in the `punct` class of the LC_CTYPE category of the current locale.

isspace
White-space character, as defined in the locale source file in the `space` class of the LC_CTYPE category of the current locale.

isupper
Uppercase letter, as defined in the locale source file in the `upper` class of the LC_CTYPE category of the current locale.

isxdigit
Hexadecimal digit (`0` through 9, `a` through `f`, or `A` through `F`), as defined in the locale source file in the `xdigit` class of the LC_CTYPE category of the current locale.

You can redefine any character class in the LC_CTYPE category of the current locale, with some restrictions. See the section about the LC_CTYPE class in the *VisualAge C++ Programming Guide* for details about these restrictions.

Returns

These functions return a nonzero value if the integer satisfies the test condition, or `0` if it does not.

Example Code

This example analyzes all characters between 0x0 and 0xFF. The output of this example is a 256-line table showing the characters from 0 to 255, indicating whether they have the properties tested for.

```
#include <stdio.h>
#include <ctype.h>
#include <locale.h>

#define UPPER_LIMIT   0xFF

int main(void)
{
   int ch;

   setlocale(LC_ALL, "En_US");
   for (ch = 0; ch <= UPPER_LIMIT; ++ch) {
      printf("%#04x ", ch);
      printf("%c", isprint(ch)  ? ch     : ' ');
      printf("%s", isalnum(ch)  ? " AN"  : "   ");
      printf("%s", isalpha(ch)  ? " A "  : "   ");
      printf("%s", iscntrl(ch)  ? " C "  : "   ");
      printf("%s", isdigit(ch)  ? " D "  : "   ");
      printf("%s", isgraph(ch)  ? " G "  : "   ");
      printf("%s", islower(ch)  ? " L "  : "   ");
      printf("%s", ispunct(ch)  ? " PU"  : "   ");
      printf("%s", isspace(ch)  ? " S "  : "   ");
      printf("%s", isprint(ch)  ? " PR"  : "   ");
      printf("%s", isupper(ch)  ? " U "  : "   ");
      printf("%s", isxdigit(ch) ? " H "  : "   ");
      putchar('\n');
   }
   return 0;



   /****************************************************************************
      The output should be similar to :
      :
      0x20                        S  PR
      0x21 !             G      PU     PR
      0x22 "             G      PU     PR
      0x23 #             G      PU     PR
      0x24 $             G      PU     PR
      0x25 %             G      PU     PR
      0x26 &             G      PU     PR
      0x27 '             G      PU     PR
      0x28 (             G      PU     PR
      0x29 )             G      PU     PR
      0x2a *             G      PU     PR
      0x2b +             G      PU     PR
      0x2c ,             G      PU     PR
      0x2d -             G      PU     PR
      0x2e .             G      PU     PR
      0x2f /             G      PU     PR
      0x30 0 AN       D  G             PR    H
      0x31 1 AN       D  G             PR    H
      0x32 2 AN       D  G             PR    H
      0x33 3 AN       D  G             PR    H
      0x34 4 AN       D  G             PR    H
      0x35 5 AN       D  G             PR    H
      0x36 6 AN       D  G             PR    H
      0x37 7 AN       D  G             PR    H
      0x38 8 AN       D  G             PR    H
      0x39 9 AN       D  G             PR    H
      :
   ****************************************************************************/
}
```

Related Information

- isascii
- iswalnum to iswxdigit
- setlocale
- tolower - toupper
- _toascii - _tolower - _toupper

----

isascii - Test Integer Values

isascii - Test Integer Values

Syntax

```
#include <ctype.h>
int isascii(int c);
```

Description

isascii tests if an integer is within the ASCII range. This macro assumes that the system uses the ASCII character set.

Note: In earlier releases of C Set ++, isascii began with an underscore (`_isascii`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_isascii` to isascii for you.

Returns

isascii returns a nonzero value if the integer is within the ASCII set, and `0` if it is not.

Example Code

This example tests the integers from `0x7c` to `0x82`, and prints the corresponding ASCII character if the integer is within the ASCII range.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   for (ch = 0x7c; ch <= 0x82; ch++) {
      printf("%#04x    ", ch);
      if (isascii(ch))
         printf("The ASCII character is %c\n", ch);
      else
         printf("Not an ASCII character\n");
   }
   return 0;

   /****************************************************************************
      The output should be:

      0x7c    The ASCII character is |
      0x7d    The ASCII character is }
      0x7e    The ASCII character is ~
      0x7f    The ASCII character is
      0x80    Not an ASCII character
      0x81    Not an ASCII character
      0x82    Not an ASCII character
   ****************************************************************************/
}
```

Related Information

- isalnum to isxdigit
- _iscsym - _iscsymf
- iswalnum to iswxdigit
- _toascii - _tolower - _toupper
- tolower - toupper

----

isatty - Test Handle for Character Device

isatty - Test Handle for Character Device

Syntax

```
#include <io.h>
int isatty(int handle);
```

Description

isatty determines whether the given handle is associated with a character device (a keyboard, display, or printer or serial port).

Note: In earlier releases of C Set ++, isatty began with an underscore (`_isatty`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_isatty` to isatty for you.

Returns

isatty returns a nonzero value if the device is a character device. Otherwise, the return value is `0`.

Example Code

This example opens the console and determines if it is a character device:

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>

int main(void)
{
   int fh,result;

   if (-1 == (fh = open("CON", O_RDWR, (S_IREAD|S_IWRITE)))) {
      perror("Error opening console\n");
      return EXIT_FAILURE;
   }
   result = isatty(fh);
   if (0 != result)
      printf("CON is a character device.\n");
   else
      printf("CON is not a character device.\n");
   close(fh);
   return 0;

   /****************************************************************************
      The output should be:

      CON is a character device.
   ****************************************************************************/
}
```

Related Information

none

_iscsym - _iscsymf - Test Integer

_iscsym - _iscsymf - Test Integer

Syntax

```
#include <ctype.h>
```

```
int _iscsym(int c);
int _iscsymf(int c);
```

These macros test if an integer is within a particular ASCII set. The macros assume that the system uses the ASCII character set.

iscsym tests if a character is alphabetic, a digit, or an underscore (). _iscsymf tests if a character is alphabetic or an underscore.

Returns

_iscsym and _iscsymf return a nonzero value if the integer is within the ASCII set for which it tests, and 0 if it is not.

Example Code

This example uses _iscsym and _iscsymf to test the characters a, _, and 1. If the character falls within the ASCII set tested for, the macro returns TRUE. Otherwise, it returns FALSE.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch[3] =  { 'a','_','1' };
   int i;

   for (i = 0; i < 3; i++) {
      printf("_iscsym('%c') returns %s\n", ch[i], _iscsym(ch[i])?"TRUE":"FALSE");
      printf("_iscsymf('%c') returns %s\n\n", ch[i], _iscsymf(ch[i])?"TRUE":
         "FALSE");
   }
   return 0;

   /****************************************************************************
      The output should be:

      _iscsym('a') returns TRUE
      _iscsymf('a') returns TRUE

      _iscsym('_') returns TRUE
      _iscsymf('_') returns TRUE

      iscsym('1') returns TRUE
      iscsymf('1') returns FALSE
   ****************************************************************************/
}
```

Related Information

- isalnum to isxdigit
- isascii
- iswalnum to iswxdigit
- tolower - toupper
- _toascii - _tolower - _toupper

----

iswalnum to iswxdigit - Test Wide Integer Value

iswalnum to iswxdigit - Test Wide Integer Value

Syntax

```
#include <wctype.h>
                                       /* test for:     */
```

```
int iswalnum(wint_t wc);  /* wide alphanumeric character */
int iswalpha(wint_t wc);  /* wide alphabetic character   */
int iswcntrl(wint_t wc);  /* wide control character      */
int iswdigit(wint_t wc);  /* wide decimal digit          */
int iswgraph(wint_t wc);  /* wide printable character, excluding space */
int iswlower(wint_t wc);  /* wide lowercase character    */
int iswprint(wint_t wc);  /* wide printable character, including space */
int iswpunct(wint_t wc);  /* wide punctuation character, excluding space */
int iswspace(wint_t wc);  /* wide whitespace character   */
int iswupper(wint_t wc);  /* wide uppercase character    */
int iswxdigit(wint_t wc); /* wide hexadecimal digit      */
```

Description

These functions test a given wide integer value *wc* to determine whether it has a certain property as defined by the LC_CTYPE category of your current locale. The value of *wc* must be representable as a `wchar_t` or WEOF.

The functions test for the following:

iswalnum
Wide alphanumeric character (uppercase or lowercase letter, or decimal digit), as defined in the locale source file in the `alnum` class of the LC_CTYPE category of the current locale.

iswalpha
Wide alphabetic character, as defined in the locale source file in the `alpha` class of the LC_CTYPE category of the current locale.

iswcntrl
Wide control character, as defined in the locale source file in the `cntrl` class of the LC_CTYPE category of the current locale.

iswdigit
Wide decimal digit (0 through 9), as defined in the locale source file in the `digit` class of the LC_CTYPE category of the current locale.

iswgraph
Wide printable character, excluding the space character, as defined in the locale source file in the `graph` class of the LC_CTYPE category of the current locale.

iswlower
Wide lowercase letter, as defined in the locale source file in the `lower` class of the LC_CTYPE category of the current locale.

iswprint
Wide printable character, including the space character, as defined in the locale source file in the `print` class of the LC_CTYPE category of the current locale.

iswpunct
Wide non-alphanumeric printable character, excluding the space character, as defined in the locale source file in the `punct` class of the LC_CTYPE category of the current locale.

iswspace
Wide white-space character, as defined in the locale source file in the `space` class of the LC_CTYPE category of the current locale.

iswupper
Wide uppercase letter, as defined in the locale source file in the `upper` class of the LC_CTYPE category of the current locale.

iswxdigit
Wide hexadecimal digit (`0` through 9, `a` through `f`, or `A` through `F`), as defined in the locale source file in the `xdigit` class of the LC_CTYPE category of the current locale.

You can redefine any character class in the LC_CTYPE category of the current locale. For more information, see "Introduction to Locales" in the *VisualAge C++ Programming Guide* .

Returns

These functions return a nonzero value if the wide integer satisfies the test value; `0` if it does not.

Example Code

This example analyzes all characters between 0x0 and 0xFF. The output of this example is a 256-line table showing the characters from 0 to 255 and indicates if the characters have the properties tested for.

```
#include <locale.h>
#include <stdio.h>
#include <wctype.h>

#define UPPER_LIMIT   0xFF

int main(void)
```

```
{
   wint_t wc;

   setlocale(LC_ALL, "En_US");

   for (wc = 0; wc <= UPPER_LIMIT; wc++) {
      printf("%#4x ", wc);
      printf("%lc", iswprint(wc) ? (wchar_t)wc    : " ");
      printf("%s", iswalnum(wc)  ? " AN" : "   ");
      printf("%s", iswalpha(wc)  ? " A " : "   ");
      printf("%s", iswcntrl(wc)  ? " C " : "   ");
      printf("%s", iswdigit(wc)  ? " D " : "   ");
      printf("%s", iswgraph(wc)  ? " G " : "   ");
      printf("%s", iswlower(wc)  ? " L " : "   ");
      printf("%s", iswpunct(wc)  ? " PU" : "   ");
      printf("%s", iswspace(wc)  ? " S " : "   ");
      printf("%s", iswprint(wc)  ? " PR" : "   ");
      printf("%s", iswupper(wc)  ? " U " : "   ");
      printf("%s", iswxdigit(wc) ? " H " : "   ");
      putchar('\n');
   }
   return 0;



   /****************************************************************************
      The output should be similar to :
      :
      0x20                        S  PR
      0x21 !             G     PU    PR
      0x22 "             G     PU    PR
      0x23 #             G     PU    PR
      0x24 $             G     PU    PR
      0x25 %             G     PU    PR
      0x26 &             G     PU    PR
      0x27 '             G     PU    PR
      0x28 (             G     PU    PR
      0x29 )             G     PU    PR
      0x2a *             G     PU    PR
      0x2b +             G     PU    PR
      0x2c ,             G     PU    PR
      0x2d -             G     PU    PR
      0x2e .             G     PU    PR
      0x2f /             G     PU    PR
      0x30 0 AN       D  G           PR    H
      0x31 1 AN       D  G           PR    H
      0x32 2 AN       D  G           PR    H
      0x33 3 AN       D  G           PR    H
      0x34 4 AN       D  G           PR    H
      0x35 5 AN       D  G           PR    H
      0x36 6 AN       D  G           PR    H
      0x37 7 AN       D  G           PR    H
      0x38 8 AN       D  G           PR    H
      0x39 9 AN       D  G           PR    H
      :
   ****************************************************************************/
}
```

Related Information

- isalnum to isxdigit
- isascii
- iswctype

----

iswctype - Test for Character Property

iswctype - Test for Character Property

Syntax

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t wc_prop);
```

Description

iswctype determines whether the wide character *wc* has the property *wc_prop* . It is similar in function to the iswalnum through isxdigit functions, but with iswctype you can specify the property to check for, or check for a property other than the standard ones.

You must obtain the *wc_prop* value from a call to wctype. If you do not, or if the LC_CTYPE category of the locale was modified after you called wctype, the behavior of iswctype is undefined.

The value of *wc* must be representable as an unsigned `wchar_t`, or WEOF.

The following strings correspond to the standard (basic) character classes or properties:
`cols='15 15 15 15'.`

```
  "
  "alnum"
  "alpha"
  "blank" "

  "
  "cntrl"
  "digit"
  "graph" "

  "
  "lower"
  "print"
  "punct" "

  "
  "space"
  "upper"
  "xdigit" "
```

The following shows calls to wctype and indicates the equivalent `isw*` function:

```
iswctype(wc, wctype("alnum"));   /* iswalnum(wc);  */
iswctype(wc, wctype("alpha"));   /* iswalpha(wc);  */
iswctype(wc, wctype("blank"));   /* iswblank(wc);  */
iswctype(wc, wctype("cntrl"));   /* iswcntrl(wc);  */
iswctype(wc, wctype("digit"));   /* iswdigit(wc);  */
iswctype(wc, wctype("graph"));   /* iswgraph(wc);  */
iswctype(wc, wctype("lower"));   /* iswlower(wc);  */
iswctype(wc, wctype("print"));   /* iswprint(wc);  */
iswctype(wc, wctype("punct"));   /* iswpunct(wc);  */
iswctype(wc, wctype("space"));   /* iswspace(wc);  */
iswctype(wc, wctype("upper"));   /* iswupper(wc);  */
iswctype(wc, wctype("xdigit"));  /* iswxdigit(wc); */
```

Returns

iswctype returns a nonzero value if the wide character has the property tested for. If the value for *wc* or *wc_prop* is not valid, the behavior is undefined.

Example Code

This example analyzes all characters between 0x0 and 0xFF. The output of this example is a 256-line table showing the characters from 0 to 255, indicating whether they have the properties tested for.

```
#include <locale.h>
#include <stdio.h>
#include <wctype.h>
```

```
#define UPPER_LIMIT   0xFF

int main(void)
{
   wint_t wc;

   setlocale(LC_ALL, "En_US");

   for (wc = 0; wc <= UPPER_LIMIT; wc++) {
      printf("%#4x ", wc);
      printf("%lc", iswctype(wc, wctype("print"))  ? (wchart)wc    : ' ');
      printf("%s", iswctype(wc, wctype("alnum"))  ? " AN" : "   ");
      printf("%s", iswctype(wc, wctype("alpha"))  ? " A " : "   ");
      printf("%s", iswctype(wc, wctype("blank"))  ? " B " : "   ");
      printf("%s", iswctype(wc, wctype("cntrl"))  ? " C " : "   ");
      printf("%s", iswctype(wc, wctype("digit"))  ? " D " : "   ");
      printf("%s", iswctype(wc, wctype("graph"))  ? " G " : "   ");
      printf("%s", iswctype(wc, wctype("lower"))  ? " L " : "   ");
      printf("%s", iswctype(wc, wctype("punct"))  ? " PU" : "   ");
      printf("%s", iswctype(wc, wctype("space"))  ? " S " : "   ");
      printf("%s", iswctype(wc, wctype("print"))  ? " PR" : "   ");
      printf("%s", iswctype(wc, wctype("upper"))  ? " U " : "   ");
      printf("%s", iswctype(wc, wctype("xdigit")) ? " H " : "   ");
      putchar('\n');
   }
   return 0;

   /****************************************************************************
      The output should be similar to :
       :
      0x1e            C
      0x1f            C
      0x20         B                 S  PR
      0x21 !                G     PU    PR
      0x22 "                G     PU    PR
      0x23 #                G     PU    PR
      0x24 $                G     PU    PR
      0x25 %                G     PU    PR
       :
      0x30 0 AN          D  G           PR    H
      0x31 1 AN          D  G           PR    H
      0x32 2 AN          D  G           PR    H
      0x33 3 AN          D  G           PR    H
      0x34 4 AN          D  G           PR    H
      0x35 5 AN          D  G           PR    H
       :
      0x43 C AN A           G           PR U  H
      0x44 D AN A           G           PR U  H
      0x45 E AN A           G           PR U  H
      0x46 F AN A           G           PR U  H
      0x47 G AN A           G           PR U
       :
   ****************************************************************************/
}
```

Related Information

- isalnum to isxdigit
- isascii
- iswalnum to iswxdigit
- iswalnum to iswxdigit
- wctype

_itoa - Convert Integer to String

_itoa - Convert Integer to String

Syntax

```
#include <stdlib.h>
char *_itoa(int value, char * string, int radix);
```

Description

_itoa converts the digits of the given *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value;* it must be in the range 2 to 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (`-`).

Note: The space reserved for *string* must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (`\0`).

Returns

_itoa returns a pointer to *string*. There is no error return value.

Example Code

This example converts the integer value -255 to a decimal, a binary, and a hex number, storing its character representation in the array `buffer`.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char buffer[35];
   char *p;

   p = _itoa(-255, buffer, 10);
   printf("The result of _itoa(-255) with radix of 10 is %s\n", p);
   p = _itoa(-255, buffer, 2);
   printf("The result of _itoa(-255) with radix of 2\n    is %s\n", p);
   p = _itoa(-255, buffer, 16);
   printf("The result of _itoa(-255) with radix of 16 is %s\n", p);
   return 0;

   /****************************************************************************
      The output should be:

      The result of _itoa(-255) with radix of 10 is -255
      The result of _itoa(-255) with radix of 2
          is 11111111111111111111111100000001
      The result of _itoa(-255) with radix of 16 is ffffff01
   ****************************************************************************/
}
```

Related Information

- _ecvt
- _fcvt
- _gcvt
- _ltoa
- _ultoa

_kbhit - Test for Keystroke

_kbhit - Test for Keystroke

Syntax

```
include <conio.h>
int _kbhit(void);
```

Description

_kbhit tests if a key has been pressed on the keyboard. If the result is nonzero, a keystroke is waiting in the buffer. You can read in the keystroke using the _getch or _getche function. If you call _getch or _getche without first calling _kbhit, the program waits for a key to be pressed.

Returns

_kbhit returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

Example Code

This example uses _kbhit to test for the pressing of a key on the keyboard and to print a statement with the test result.

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
   int ch;

   printf("Type in some letters.\n");
   printf("If you type in an 'x', the program ends.\n");

   for (; ; ) {
      while (0==_kbhit()){
      /* Processing without waiting for a key to be pressed */
      }

      ch = getch();
      printf("You have pressed the '%c' key.\n",ch);

      if ('x' == ch)
         break;
   }
   return 0;

   /**************************************************************************
      The output should be similar to:

      Type in some letters.
      If you type in an 'x', the program ends.
      You have pressed the 'f' key.
      You have pressed the 'e' key.
      You have pressed the 'l' key.
      You have pressed the 'i' key.
      You have pressed the 'x' key.
   **************************************************************************/
}
```

Related Information

- _getch - _getche

----

labs - Calculate Absolute Value of Long Integer

labs - Calculate Absolute Value of Long Integer

Syntax

```
#include <stdlib.h>
long int labs(long int n);
```

Description

`labs` produces the absolute value of its long integer argument *n*. The result may be undefined when the argument is equal to `LONG_MIN`, the smallest available long integer (-2 147 483 647). The value `LONG_MIN` is defined in the `<limits.h>` include file.

Returns

`labs` returns the absolute value of *n*. There is no error return value.

Example Code

This example computes *y* as the absolute value of the long integer -41567.

```
#include <stdlib.h>

int main(void)
{
   long x,y;

   x = -41567L;
   y = labs(x);
   printf("The absolute value of %ld is %ld\n", x, y);
   return 0;

   /*****************************************************************************
      The output should be:

      The absolute value of -41567 is 41567
   *****************************************************************************/
}
```

Related Information

- abs
- fabs

----

ldexp - Multiply by a Power of Two

ldexp - Multiply by a Power of Two

Syntax

```
#include <math.h>
double ldexp(double x, int exp);
```

Description

`ldexp` calculates the value of $x * (2^{exp})$.

Returns

`ldexp` returns the value of $x*(2^{exp})$. If an overflow results, the function returns `+HUGE_VAL` for a large result or

`-HUGE_VAL` for a small result, and sets `errno` to `ERANGE`.

Example Code

This example computes *y* as 1.5 times 2 to the fifth power (1.5*25):

```
#include <math.h>

int main(void)
{
   double x,y;
   int p;

   x = 1.5;
   p = 5;
   y = ldexp(x, p);
   printf("%lf times 2 to the power of %d is %lf\n", x, p, y);
   return 0;

   /***************************************************************************
      The output should be:

      1.500000 times 2 to the power of 5 is 48.000000
   ***************************************************************************/
}
```

Related Information

- exp
- frexp
- modf

---

ldiv - Perform Long Division

ldiv - Perform Long Division

Syntax

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

Description

`ldiv` calculates the quotient and remainder of the division of *numerator* by *denominator* .

Returns

`ldiv` returns a structure of type `ldiv_t`, containing both the quotient (`long int quot`) and the remainder (`long int rem`). If the value cannot be represented, the return value is undefined. If *denominator* is `0`, an exception is raised.

Example Code

This example uses `ldiv` to calculate the quotients and remainders for a set of two dividends and two divisors.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
```

```
{
   long int num[2] =  { 45,-45 };
   long int den[2] =  { 7,-7 };
   ldiv_t ans;       /* ldiv_t is a struct type containing two long ints:
                        'quot' stores quotient; 'rem' stores remainder  */
   short i,j;

   printf("Results of long division:\n");
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++) {
         ans = ldiv(num[i], den[j]);
         printf("Dividend: %6ld  Divisor: %6ld", num[i], den[j]);
         printf("  Quotient: %6ld  Remainder: %6ld\n", ans.quot, ans.rem);
      }
   return 0;

   /****************************************************************************
      The output should be:

      Results of long division:
      Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
      Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
      Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
      Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3
   ****************************************************************************/
}
```

Related Information

- div

----

lfind - lsearch - Find Key in Array

lfind - lsearch - Find Key in Array

Syntax

```
#include <search.h>
char *lfind(char *search_key, char *base,
             unsigned int *num, unsigned int *width,
             int (*compare)(const void *key, const void *element));
char *lsearch(char *search_key, char *base,
               unsigned int *num, unsigned int *width,
               int (*compare)(const void *key, const void *element));
```

Description

lfind and lsearch perform a linear search for the value *search_key* in an array of *num* elements, each of *width* bytes in size. Unlike `bsearch`, lsearch and lfind do not require that you sort the array first. The argument *base* points to the base of the array to be searched.

If lsearch does not find the *search_key*, it adds the *search_key* to the end of the array and increments *num* by one. If lfind does not find the *search_key*, it does not add the *search_key* to the array.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. Both lfind and lsearch call this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values:

Value	**Meaning**
Nonzero	*key* and *element* are different.
`0`	*key* and *element* are identical.

Note: In earlier releases of C Set ++, lfind and lsearch began with an underscore (`_lfind` and `_lsearch`). Because they are defined by the X/Open standard, the underscore has been removed. For compatibility, *The*

Developer's Toolkit will map `_lfind` and `_lsearch` to lfind and lsearch for you.

Returns

If *search_key* is found, both lsearch and lfind return a pointer to that element of the array to which *base* points. If *search_key* is not found, lsearch returns a pointer to a newly added item at the end of the array, while lfind returns `NULL`.

Example Code

This example uses lfind to search for the keyword PATH in the command-line arguments.

```
#include <search.h>
#include <string.h>
#include <stdio.h>

#define  CNT           2

int compare(const void *arg1,const void *arg2)
{
   return (strncmp(*(char **)arg1, *(char **)arg2, strlen(*(char **)arg1)));
}

int main(void)
{
   char **result;
   char *key = "PATH";
   unsigned int num = CNT;
   char *string[CNT] =  {
      "PATH = d:\\david\\matthew\\heather\\ed\\simon","LIB = PATH\\abc" };

   /* The following statement finds the argument that starts with "PATH"      */

   if ((result = (char **)lfind((char *)&key, (char *)string, &num,
                  sizeof(char *), compare)) != NULL)
      printf("%s found\n", *result);
   else
      printf("PATH not found \n");
   return 0;

   /****************************************************************************
      The output should be:

      PATH = d:\david\matthew\heather\ed\simon found
   ****************************************************************************/
}
```

Related Information

- bsearch
- qsort

----

localeconv - Retrieve Information from the Environment

localeconv - Retrieve Information from the Environment

Syntax

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

localeconv retrieves information about the environment for the current locale and places the information in a structure of type struct lconv. Subsequent calls to localeconv, or to setlocale with the argument LC_ALL, LC_MONETARY, or LC_NUMERIC, can overwrite the structure.

The structure contains the members listed below. Pointers to strings with a value of "" indicate that the value is not available in this locale or is of zero length. Character types with a value of CHAR_MAX indicate that the value is not available in this locale.

Element	Purpose of Element
"char *decimal_point"	Decimal-point character for formatting nonmonetary quantities.
"char *thousands_sep"	Character used to separate groups of digits to the left of the decimal-point character in formatted nonmonetary quantities.
"char *grouping"	Size of each group of digits in formatted nonmonetary quantities. The value of each character in the string determines the number of digits in a group. "CHAR_MAX" indicates that there are no further groupings. If the last integer is not "CHAR_MAX", then the size of the previous group will be used for the remainder of the digits.
"char *int_curr_symbol"	International currency symbol. The first three characters contain the alphabetic international currency symbol. The fourth character (usually a space) separates the international currency symbol from the monetary quantity.
"char *currency_symbol"	Local currency symbol.
"char *mon_decimal_point"	Decimal-point character for formatting monetary quantities.
"char *mon_thousands_sep"	Separator for digits in formatted monetary quantities.
"char *mon_grouping"	Size of each group of digits in formatted monetary quantities. The value of each character in the string determines the number of digits in a group. "CHAR_MAX" indicates that there are no further groupings. If the last integer is not "CHAR_MAX", then the size of the previous group will be used for the remainder of the digits.
"char *positive_sign"	Positive sign used in monetary quantities.
"char *negative_sign"	Negative sign used in monetary quantities.
"char int_frac_digits"	Mumber of displayed digits to the right of the decimal place for internationally formatted monetary quantities.
"char frac_digits"	Number of digits to the right of the decimal place in monetary quantities.

"char p_cs_precedes" 1 if the "currency_symbol" precedes the value for a nonnegative formatted monetary quantity; "0" if it does not.

"char p_sep_by_space" 1 if the "currency_symbol" is separated by a space from the value of a nonnegative formatted monetary quantity; "0" if it is not.

"char n_cs_precedes" 1 if the "currency_symbol" precedes the value for a negative formatted monetary quantity; "0" if it does not.

"char n_sep_by_space" 1 if the "currency_symbol" is separated by a space from the value of a negative formatted monetary quantity; "0" if it is not.

"char p_sign_posn" Position of the "positive_sign" for a nonnegative formatted monetary quantity.

"char n_sign_posn" Position of the "negative_sign" for a negative formatted monetary quantity.

"char *left_parenthesis" Symbol to appear to the left of a negative-valued monetary symbol (such as a loss or deficit).

"char *right_parenthesis" Symbol to appear to the right of a negative-valued monetary symbol (such as a loss or deficit).

"char *debit_sign" String to indicate a non-negative-valued formatted monetary quantity.

"char *credit_sign" String to indicate a negative-valued formatted monetary quantity.

The `grouping` and `mon_grouping` members can have the following values:

Value	**Meaning**
CHAR_MAX	No further grouping is to be performed.
`0`	The previous element is to be repeatedly used for the rest of the digits.
other	The number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The `n_sign_posn` and `p_sign_posn` elements can have the following values:

Value	**Meaning**
`0`	The quantity and `currency_symbol` are enclosed in parentheses.
`1`	The sign precedes the quantity and `currency_symbol`.
`2`	The sign follows the quantity and `currency_symbol`.
`3`	The sign precedes the `currency_symbol`.
`4`	The sign follows the `currency_symbol`.

Returns

`localeconv` returns a pointer to the `lconv` structure. Calls to setlocale with the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

Example Code

This example prints out the default decimal point for your locale and then the decimal point for the `Fr_FR.IBM-850` locale.

```
#include <stdio.h>
#include <locale.h>

int main(void)
{
   struct lconv *mylocale;

   mylocale = localeconv();
   printf("Default decimal point is a %s\n", mylocale->decimal_point);

   if (NULL != setlocale(LC_ALL, Fr_FR.IBM-850)) {
      mylocale = localeconv();
      printf("France's decimal point is a %s\n", mylocale->decimal_point);
   } else {
      printf("setlocale(LC_ALL, Fr_FR.IBM-850) returned <NULL>\n");
    }
   return 0;

   /****************************************************************************
      The output should be similar to :

      Default decimal point is a .
      France's decimal point is a ,
   ****************************************************************************/
}
```

Related Information

- setlocale

----

localtime - Convert Time

localtime - Convert Time

Syntax

```
#include <time.h>
struct tm *localtime(const time_t *timeval);
```

Description

`localtime` breaks down *timeval*, corrects for the local time zone and Daylight Saving Time, if appropriate, and stores the corrected time in a structure of type `tm`. See gmtime for a description of the fields in a `tm` structure.

The time value is usually obtained by a call to the `time` function.

Note:

- `gmtime` and `localtime` may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
- The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Returns

`localtime` returns a pointer to the structure result. If unsuccessful, it returns NULL.

Example Code

This example queries the system clock and displays the local time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   struct tm *newtime;
   time_t ltime;

   time(&ltime);
   newtime = localtime(&ltime);
   printf("The date and time is %s", asctime(newtime));
   return 0;

   /****************************************************************************
      The output should be similar to:

      The date and time is Wed Oct 31 15:00:00 1995
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- mktime
- time

----

log - Calculate Natural Logarithm

log - Calculate Natural Logarithm

Syntax

```
#include <math.h>
double log(double x);
```

Description

`log` calculates the natural logarithm (base e) of *x*.

Returns

`log` returns the computed value. If *x* is negative, log sets `errno` to `EDOM` and may return the value `-HUGE_VAL`. If *x* is zero, `log` returns the value `-HUGE_VAL`, and may set `errno` to `ERANGE`.

Example Code

This example calculates the natural logarithm of 1000.0.

```
#include <math.h>

int main(void)
{
   double x = 1000.0,y;

   y = log(x);
   printf("The natural logarithm of %lf is %lf\n", x, y);
   return 0;
   /************************************************************************
      The output should be:

      The natural logarithm of 1000.000000 is 6.907755
   ************************************************************************/
}
```

Related Information

- exp
- log10
- pow

----

log10 - Calculate Base 10 Logarithm

log10 - Calculate Base 10 Logarithm

Syntax

```
#include <math.h>
double log10(double x);
```

Description

`log10` calculates the base 10 logarithm of *x*.

Returns

`log10` returns the computed value. If *x* is negative, log10 sets `errno` to `EDOM` and may return the value `-HUGE_VAL`. If *x* is zero, `log10` returns the value `-HUGE_VAL`, and may set `errno` to `ERANGE`.

Example Code

This example calculates the base 10 logarithm of 1000.0.

```
#include <math.h>

int main(void)
{
   double x = 1000.0,y;

   y = log10(x);
   printf("The base 10 logarithm of %lf is %lf\n", x, y);
   return 0;

   /************************************************************************
      The output should be:

      The base 10 logarithm of 1000.000000 is 3.000000
   ************************************************************************/
}
```

Related Information

- exp
- log
- pow

----

longjmp - Restore Stack Environment

longjmp - Restore Stack Environment

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

Description

longjmp restores a stack environment previously saved in *env* by setjmp. setjmp and longjmp provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to setjmp causes the current stack environment to be saved in *env*. A subsequent call to longjmp restores the saved environment and returns control to a point in the program corresponding to the setjmp call. Execution resumes as if the setjmp call had just returned the given *value*.

All variables (except register variables) that are accessible to the function that receives control contain the values they had when longjmp was called. The values of variables that are allocated to registers by the compiler are unpredictable. Because any auto variable could be allocated to a register in optimized code, you should consider the values of all auto variables to be unpredictable after a longjmp call.

Note: Ensure that the function that calls setjmp does not return before you call the corresponding longjmp function. Calling longjmp after the function calling setjmp returns causes unpredictable program behavior.

The *value* argument must be nonzero. If you give a zero argument for *value*, longjmp substitutes 1 in its place.

C++ Considerations When you call setjmp in a C++ program, ensure that that part of the program does not also create C++ objects that need to be destroyed. When you call longjmp, objects existing at the time of the setjmp call will still exist, but any destructors called after setjmp are not called. For example, given the following program:

```
#include <stdio.h>
#include <setjmp.h>

class A {
   int i;
   public:
      A(int I) {i = I; printf("Constructed at line %d\n", i);}
      ~A() {printf("Destroying object constructed at line %d\n",i);}
   };
jmp_buf jBuf;

int main(void) {
   A a1(__LINE__);
   if (!setjmp(jBuf)) {
     A a2(__LINE__);
     printf("Press y (and Enter) to longjmp; anything else to return norm
     char c = getchar();
     if (c == 'y')
        longjmp(jBuf, 1);
   }
   A a3(__LINE__);
   return 0;
}
```

If you return normally, the output would be:

```
Constructed at line 13
Constructed at line 15
Press y (and Enter) to longjmp; anything else to return normally
x
Destroying object constructed at line 15
Constructed at line 21
Destroying object constructed at line 21
Destroying object constructed at line 13
```

If you called longjmp, the output would be:

```
Constructed at line 13
Constructed at line 15
Press y (and Enter) to longjmp; anything else to return normally
y
Constructed at line 21
Destroying object constructed at line 21
Destroying object constructed at line 13
```

Notice that the object from line 15 is not destroyed.

Returns

`longjmp` does not use the normal function call and return mechanisms; it has no return value.

Example Code

This example saves the stack environment at the statement:

```
   if(setjmp(mark) != 0) ...
```

When the system first performs the `if` statement, it saves the environment in `mark` and sets the condition to `FALSE` because `setjmp` returns a 0 when it saves the environment. The program prints the message:

```
   setjmp has been called
```

The subsequent call to function `p` tests for a local error condition, which can cause it to call `longjmp`. Then, control returns to the original `setjmp` function using the environment saved in `mark`. This time, the condition is `TRUE` because -1 is the return value from `longjmp`. The example then performs the statements in the block, prints `longjmp has been called`, calls `recover`, and leaves the program.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

int main(void)
{
   if (setjmp(mark) != 0) {
      printf("longjmp has been called\n");
      recover();
      exit(1);
   }
   printf("setjmp has been called\n");

   p();

   return 0;

   /*****************************************************************************
      The output should be:

      setjmp has been called
      longjmp has been called
   ****************************************************************************/
}

int p(void)
{
   int error = 0;
   error = 9;

   if (error != 0)
      longjmp(mark, -1);

}

int recover(void)
{
}
```

Related Information

- setjmp
- goto in the *Language Reference*

----

lseek - Move File Pointer

lseek - Move File Pointer

Syntax

```
#include <io.h>  /* constants in <stdio.h> */
long lseek(int handle, long offset, int origin);
```

Description

lseek moves any file pointer associated with *handle* to a new location that is *offset* bytes from the *origin* . The next operation on the file takes place at the new location. The *origin* must be one of the following constants, defined in `<stdio.h>`: compact break=fit.

Origin	**Definition**
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file.

lseek can reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file; the data between the EOF and the new file position is undefined. (See chsize for more information on extending the length of the file.) An attempt to position the pointer before the beginning of the file causes an error.

Note: In earlier releases of C Set ++, lseek began with an underscore (`_lseek`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_lseek` to lseek for you.

Returns

lseek returns the offset, in bytes, of the new position from the beginning of the file. A return value of `-1L` indicates an error, and lseek sets `errno` to one of the following values: compact break=fit.

Value	**Meaning**
EBADF	The file handle is incorrect.
EINVAL	The value for *origin* is incorrect, or the position specified by *offset* is before the beginning of the file.
EOS2ERR	The call to the operating system was not successful.

On devices incapable of seeking (such as keyboards and printers), lseek returns `-1` and sets `errno` to EOS2ERR.

Example Code

This example opens the file `sample.dat` and, if successful, moves the file pointer to the eighth position in the file. The example then attempts to read bytes from the file, starting at the new pointer position, and reads them into the buffer.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
   long length;
   int fh;
   char buffer[20];

   memset(buffer, '\0', 20);                /* Initialize the buffer           */
   printf("\nCreating sample.dat.\n");
   system("echo Sample Program > sample.dat");
   if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
      perror("Unable to open sample.dat.");
      exit(EXIT_FAILURE);
   }
   if (-1 == lseek(fh, 7, SEEK_SET)) {      /* place the file pointer at the   */
      perror("Unable to lseek");            /* eighth position in the file     */
      close(fh);
      return EXIT_FAILURE;
   }
   if (8 != read(fh, buffer, 8)) {
      perror("Unable to read from sample.dat.");
      close(fh);
      return EXIT_FAILURE;
   }
   printf("Successfully read in the following:\n%s.\n", buffer);
   close(fh);
   return 0;

   /*****************************************************************************
      The output should be:

      Creating sample.dat.
      Successfully read in the following:
      Program .
   ****************************************************************************/
}
```

Related Information

- chsize
- fseek
- _tell

_ltoa - Convert Long Integer to String

_ltoa - Convert Long Integer to String

Syntax

```
#include <stdlib.h>
char *_ltoa(long value, char *string, int radix);
```

Description

_ltoa converts the digits of the given long integer *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2 to 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (`-`).

Note: The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes, including the null character (`\0`).

Returns

_ltoa returns a pointer to *string*. There is no error return value.

Example Code

This example converts the long integer -255L to a decimal, binary, and hex value, and stores its character representation in the array `buffer`.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char buffer[35];
   char *p;

   p = _ltoa(-255L, buffer, 10);
   printf("The result of _ltoa(-255) with radix of 10 is %s\n", p);
   p = _ltoa(-255L, buffer, 2);
   printf("The result of _ltoa(-255) with radix of 2\n    is %s\n", p);
   p = _ltoa(-255L, buffer, 16);
   printf("The result of _ltoa(-255) with radix of 16 is %s\n", p);
   return 0;

   /****************************************************************************
      The output should be:

      The result of _ltoa(-255) with radix of 10 is -255
      The result of _ltoa(-255) with radix of 2
          is 11111111111111111111111100000001
      The result of _ltoa(-255) with radix of 16 is ffffff01
   ****************************************************************************/
}
```

Related Information

- atol
- _ecvt
- _fcvt
- _gcvt
- _itoa
- strtol
- _ultoa
- wcstol

_makepath - Create Path

_makepath - Create Path

Syntax

```
#include <stdlib.h>
```

```
void _makepath(char *path, char *drive, char *dir, char *fname, char *ext);
```

Description

_makepath creates a single path name, composed of a drive letter, directory path, file name, and file name extension.

The *path* argument should point to an empty buffer large enough to hold the complete path name. The constant _MAX_PATH, defined in `<stdlib.h>`, specifies the maximum size allowed for *path* . The other arguments point to the following buffers containing the path name elements:

drive A letter (A, B, ...) corresponding to the desired drive and an optional following colon. _makepath inserts the colon automatically in the composite path name if it is missing. If *drive* is a null character or an empty string, no drive letter or colon appears in the composite *path* string.

dir The path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and either slash (/) or backslash (\) or both can be used in a single *dir* argument. If a trailing slash or backslash is not specified, it is inserted automatically. If *dir* is a null character or an empty string, no slash is inserted in the composite *path* string.

fname The base file name without any extensions.

ext The actual file name extension, with or without a leading period. _makepath inserts the period automatically if it does not appear in *ext* . If *ext* is a null character or an empty string, no period is inserted in the composite *path* string.

The size limits on the above four fields are those specified by the constants _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT, which are defined in `<stdlib.h>`. The composite path should be no larger than the _MAX_PATH constant also defined in `<stdlib.h>`; otherwise, the operating system does not handle it correctly.

Note: No checking is done to see if the syntax of the file name is correct.

Returns

There is no return value.

Example Code

This example builds a file name path from the specified components:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath(path_buffer, "c", "qc\\bob\\eclibref\\e", "makepath", "c");
   printf("Path created with _makepath: %s\n\n", path_buffer);
   _splitpath(path_buffer, drive, dir, fname, ext);
   printf("Path extracted with _splitpath:\n");
   printf("drive: %s\n", drive);
   printf("directory: %s\n", dir);
   printf("file name: %s\n", fname);
   printf("extension: %s\n", ext);
   return 0;

   /****************************************************************************
      The output should be:

      Path created with _makepath: c:qc\bob\eclibref\e\makepath.c

      Path extracted with _splitpath:
      drive: c:
      directory: qc\bob\eclibref\e\
      file name: makepath
      extension: .c
   ****************************************************************************/
}
```

Related Information

- _fullpath
- _splitpath

----

malloc - Reserve Storage Block

malloc - Reserve Storage Block

Syntax

```
#include <stdlib.h>  /* also in <malloc.h> */
void *malloc(size_t size);
```

Description

`malloc` reserves a block of storage of *size* bytes. Unlike `calloc`, `malloc` does not initialize all elements to `0`.

Heap-specific and debug versions of this function (_umalloc and _debug_malloc) are also available. malloc always operates on the default heap. For more information about memory management, see "Managing Memory" in the *VisualAge C++ Programming Guide* .

Returns

`malloc` returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. The return value is `NULL` if not enough storage is available, or if *size* was specified as zero.

Example Code

This example prompts for the number of array entries you want and then reserves enough space in storage for the entries. If `malloc` was successful, the example assigns values to the entries and prints out each entry; otherwise, it prints out an error.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long *array;                              /* start of the array            */
   long *index;                                           /* index variable   */
   int i;                                                 /* index variable   */
   int num;                             /* number of entries of the array     */

   printf("Enter the size of the array\n");
   scanf("%i", &num);

   /* allocate num entries                                                    */

   if ((index = array = malloc(num *sizeof(long))) != NULL) {
      for (i = 0; i < num; ++i)             /* put values in array            */
         *index++ = i;                      /* using pointer notation         */
      for (i = 0; i < num; ++i)             /* print the array out            */
         printf("array[ %i ] = %i\n", i, array[i]);
   }
   else {                                                 /* malloc error     */
      perror("Out of storage");
      abort();
   }
   return 0;
   /****************************************************************************
      The output should be similar to:

      Enter the size of the array
      5
      array[ 0 ] = 0
      array[ 1 ] = 1
      array[ 2 ] = 2
      array[ 3 ] = 3
      array[ 4 ] = 4
   ****************************************************************************/
}
```

Related Information

- calloc
- free
- _mheap
- _msize
- realloc
- _umalloc

_matherr - Process Math Library Errors

_matherr - Process Math Library Errors

Syntax

```
#include <math.h>
int _matherr(struct exception *x);
```

Description

_matherr processes errors generated by the functions in the math library. The math functions call _matherr whenever they detect an error. The _matherr function supplied with *The Developer's Toolkit* library performs no error handling and returns `0` to the calling function.

You can provide a different definition of _matherr to carry out special error handling. Be sure to use the `/NOE` link option, if you are using your own _matherr function. For *The Developer's Toolkit* compiler, you can use the `/B` option on the icc command line to pass the `/NOE` option to the linker, as in the following:

```
   icc /B"/NOE" yourfile.c
```

When an error occurs in a math routine, _matherr is called with a pointer to the following structure (defined in `<math.h>`) as an argument:

```
   struct exception {
       int type;
       char *name;
       double arg1, arg2, retval;
   };
```

The `type` field specifies the type of math error. It has one of the following values, defined in `<math.h>`: compact break=fit.

Value	**Meaning**
DOMAIN	Argument domain error

OVERFLOW	Overflow range error
UNDERFLOW	Underflow range error
TLOSS	Total loss of significance
PLOSS	Partial loss of significance
SING	Argument singularity.

PLOSS is provided for compatibility with other compilers; *The Developer's Toolkit* does not generate this value.

The `name` is a pointer to a null-ended string containing the name of the function that caused the error. The `arg1` and `arg2` fields specify the argument values that caused the error. If only one argument is given, it is stored in `arg1`. The `retval` is the default return value; you can change it.

Returns

The return value from _matherr must specify whether or not an error actually occurred. If _matherr returns `0`, an error message appears, and errno is set to an appropriate error value. If _matherr returns a nonzero value, no error message appears and errno remains unchanged.

Example Code

This example provides a _matherr function to handle errors from the `log` or `log10` functions. The arguments to these logarithmic functions must be positive double values. _matherr processes a negative value in an argument (a domain error) by returning the log of its absolute value. It suppresses the error message normally displayed when this error occurs. If the error is a zero argument or if some other routine produced the error, the example takes the default actions.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int value;

   printf("Trying to evaluate log10(-1000.00) will create a math exception.\n");
   value = log10(-1000.00);
   printf("The _matherr() exception handler evaluates the expression to\n");
   printf("log10(-1000.00) = %d\n", value);
   return 0;

   /****************************************************************************
      The output should be:

      Trying to evaluate log10(-1000.00) will create a math exception.
      inside _matherr
      The _matherr() exception handler evaluates the expression to
      log10(-1000.00) = 3
   ****************************************************************************/
}

int _matherr(struct exception *x)
{
   printf("inside _matherr\n");
   if (DOMAIN == x->type) {
      if (0 == strcmp(x->name, "log")) {
         x->retval = log(-(x->arg1));
         return EXIT_FAILURE;
      }
      else
         if (0 == strcmp(x->name, "log10")) {
            x->retval = log10(-(x->arg1));
            return EXIT_FAILURE;
         }
   }
   return 0;                                  /* Use default actions          */
}
```

Related Information

- log
- log10

----

max - Return Larger of Two Values

max - Return Larger of Two Values

Syntax

```
#include <stdlib.h>
type max(type a, type b);
```

Description

`max` compares two values and determines the larger of the two. The data *type* can be any arithmetic data type, signed or unsigned. Both arguments must have the same type for each call to max.

Note: Because `max` is a macro, if the evaluation of the arguments contains side effects (post-increment operators, for example), the results of both the side effects and the macro will be undefined.

Returns

`max` returns the larger of the two values.

Example Code

This example prints the larger of the two values, `a` and `b`.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int a = 10;
   int b = 21;

   printf("The larger of %d and %d is %d\n", a, b, max(a, b));
   return 0;

   /****************************************************************************
      The output should be:

      The larger of 10 and 21 is 21
   ****************************************************************************/
}
```

Related Information

- min

----

mblen - Determine Length of Multibyte Character

mblen - Determine Length of Multibyte Character

Syntax

```
#include <stdlib.h>
int mblen(const char *string, size_t n);
```

Description

mblen determines the length in bytes of the multibyte character pointed to by *string*. A maximum of *n* bytes is examined.

The behavior of mblen is affected by the LC_CTYPE category of the current locale.

Returns

If *string* is `NULL`, mblen returns `0`.

If *string* is not `NULL`, mblen returns:

- `0` if *string* points to the null character
- The number of bytes comprising the multibyte character
- The value -1 if *string* does not point to a valid multibyte character.

Example Code

This example uses mblen and mbtowc to convert a multibyte character into a single wide character.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

int main(void)
{
   char     mb_string[] = "\x81\x41\x81\xc2" "b";
   int      length;
   wchar_t  widechar;

   if (NULL == setlocale(LC_ALL, "ja_jp.ibm-932")) {
      printf("setlocale failed.\n");
      exit(EXIT_FAILURE);
   }
   length = mblen(mb_string, MB_CUR_MAX);
   mbtowc(&widechar, mb_string, length);
   printf("The wide character %lc has length of %d.\n", widechar, length);
   return 0;

   /****************************************************************************
      The output should be similar to :

      The wide character ×A has length of 2.
   ****************************************************************************/
}
```

Related Information

- mbstowcs
- mbtowc
- setlocale
- strlen
- wcslen
- wctomb

----

mbstowcs - Convert Multibyte String to Wide-Character String

mbstowcs - Convert Multibyte String to Wide-Character String

Syntax

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *dest, const char *string, size_t len);
```

Description

mbstowcs converts the multibyte character string pointed to by *string* into the wide-character array pointed to by *dest*. Depending on the encoding scheme used by the code set, the multibyte character string can contain any combination of single-byte or double-byte characters.

The conversion stops after *len* wide-characters in *dest* are filled or after a null byte is encountered. The terminating null character is converted to a wide character with the value `0`; characters that follow it are not processed.

The behavior of mbstowcs is affected by the LC_CTYPE category of the current locale.

Returns

If successful, mbstowcs returns the number of characters converted and stored in *dest*, not counting the terminating null character. The string pointed to by *dest* ends with a null character unless mbstowcs returns the value *len*.

If it encounters an invalid multibyte character, mbstowcs returns `(size_t)-1`. If *dest* is a null pointer, the value of *len* is ignored and mbstowcs returns the number of wide characters required for the converted multibyte characters.

Example Code

This example uses mbstowcs to convert the multibyte character `mbs` to a wide character string and store it in `wcs`.

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define SIZE 10

int main(void)
{
   char        mbs[] = "\x81\x41" "m" "\x81\x42";
   wchar_t     wcs[SIZE];

   if (NULL == setlocale(LC_ALL, "ja_jp.ibm-932")) {
      printf("setlocale failed.\n");
      exit(EXIT_FAILURE);
   }
   mbstowcs(wcs, mbs, SIZE);
   printf("The wide character string is %ls.\n", wcs);
   return 0;

   /****************************************************************************
      The output should be similiar to :

      The wide character string is ×Am×B.
   ****************************************************************************/
}
```

Related Information

- mblen
- mbtowc
- setlocale
- wcslen
- wcstombs

----

mbtowc - Convert Multibyte Character to Wide Character

mbtowc - Convert Multibyte Character to Wide Character

Syntax

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

Description

mbtowc first determines the length of the multibyte character pointed to by *string*. It then converts the multibyte character to the corresponding wide character, and stores it in the location pointed to by *pwc*, if *pwc* is not a null pointer. mbtowc examines a maximum of *n* bytes from *string*.

If *pwc* is a null pointer, the multibyte character is not converted.

The behavior of mbtowc is affected by the LC_CTYPE category of the current locale.

Returns

If *string* is `NULL`, mbtowc returns `0`. If *string* is not `NULL`, mbtowc returns:

- The number of bytes comprising the converted multibyte character, if *n* or fewer bytes form a valid multibyte character.
- 0 if *string* points to the null character.
- -1 if *string* does not point to a valid multibyte character, and the next *n* bytes do not form a valid multibyte character.

Example Code

This example uses mbtowc to convert the second multibyte character in `mbs` to a wide character.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

int main(void)
{
   char     mb_string[] = "\x81\x41\x81\x42" "c" "\x00";
   int      length;
   wchar_t  widechar;

   if (NULL == setlocale(LC_ALL, "ja_jp.ibm-932")) {
      printf("setlocale failed.\n");
      exit(EXIT_FAILURE);
   }
   length = mblen(mb_string, MB_CUR_MAX);
   length = mbtowc(&widechar, mb_string + length, MB_CUR_MAX);
   printf("The wide character %lc has length of %d.\n", widechar, length);
   return 0;

   /****************************************************************************
      The output should be similar to :

      The wide character ×B has length of 2.
   ****************************************************************************/
}
```

Related Information

- mblen

- mbstowcs
- setlocale
- wcslen
- wctomb

----

memchr - Search Buffer

memchr - Search Buffer

Syntax

```
#include <string.h>  /* also in <memory.h> */
void *memchr(const void *buf, int c, size_t count);
```

Description

`memchr` searches the first *count* bytes of *buf* for the first occurrence of *c* converted to an unsigned *char*. The search continues until it finds *c* or examines *count* bytes.

Returns

`memchr` returns a pointer to the location of *c* in *buf*. It returns `NULL` if *c* is not within the first *count* bytes of *buf*.

Example Code

This example finds the first occurrence of "x" in the string that you provide. If it is found, the string that starts with that character is printed.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   char *result;

   if (argc != 2)
      printf("Usage: %s string\n", argv[0]);
   else {
      if ((result = memchr(argv[1], 'x', strlen(argv[1]))) != NULL)
         printf("The string starting with x is %s\n", result);
      else
         printf("The letter x cannot be found in the string\n");
   }
   return 0;

   /****************************************************************************
      If the program is passed the argumrnt boxing, the output should be:

      The string starting with x is xing
   ****************************************************************************/
}
```

Related Information

- memcmp
- memcpy
- memicmp
- memmove
- memset
- strchr

----

memcmp - Compare Buffers

memcmp - Compare Buffers

Syntax

```
#include <string.h>  /* also in <memory.h> */
int memcmp(const void *buf1, const void *buf2, size_t count);
```

Description

`memcmp` compares the first *count* bytes of *buf1* and *buf2*.

Returns

`memcmp` returns a value indicating the relationship between the two buffers as follows: compact break=fit.

Value	Meaning
Less than `0`	*buf1* less than *buf2*
`0`	*buf1* identical to *buf2*
Greater than `0`	*buf1* greater than *buf2*

Example Code

This example compares first and second arguments passed to `main` to determine which, if either, is greater.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   int len;
   int result;

   if (argc != 3) {
      printf("Usage: %s string1 string2\n", argv[0]);
   }
   else {

      /* Determine the length to be used for comparison                      */

      if (strlen(argv[1]) < strlen(argv[2]))
         len = strlen(argv[1]);
      else
         len = strlen(argv[2]);
      result = memcmp(argv[1], argv[2], len);
      printf("When the first %i characters are compared,\n", len);
      if (0 == result)
         printf("\"%s\" is identical to \"%s\"\n", argv[1], argv[2]);
      else
         if (result < 0)

            printf("\"%s\" is less than \"%s\"\n", argv[1], argv[2]);
         else
            printf("\"%s\" is greater than \"%s\"\n", argv[1], argv[2]);
   }
   return 0;

   /****************************************************************************
      If the program is passed the arguments "firststring secondstring",
      the output should be:
```

```
      When the first 11 characters are compared,
      "firststring" is less than "secondstring"
   ****************************************************************************/
}
```

Related Information

- memchr
- memcpy
- memicmp
- memmove
- memset
- strcmp

----

memcpy - Copy Bytes

memcpy - Copy Bytes

Syntax

```
#include <string.h>  /* also <memory.h> */
void *memcpy(void *dest, const void *src, size_t count);
```

Description

`memcpy` copies *count* bytes of *src* to *dest*. The behavior is undefined if copying takes place between objects that overlap. (The `memmove` function allows copying between objects that may overlap.)

Returns

`memcpy` returns a pointer to *dest*.

Example Code

This example copies the contents of `source` to `target`.

```
#include <string.h>
#include <stdio.h>

#define  MAX_LEN       80
char source[MAX_LEN] = "This is the source string";
char target[MAX_LEN] = "This is the target string";

int main(void)
{
   printf("Before memcpy, target is \"%s\"\n", target);
   memcpy(target, source, sizeof(source));
   printf("After memcpy, target becomes \"%s\"\n", target);
   return 0;

   /****************************************************************************
      The output should be:

      Before memcpy, target is "This is the target string"
      After memcpy, target becomes "This is the source string"
   ****************************************************************************/
}
```

Related Information

- memchr
- memcmp
- memmove
- memset
- strcpy

----

memicmp - Compare Bytes

memicmp - Compare Bytes

Syntax

```
#include <string.h>    /* also in <memory.h> */
int memicmp(void *buf1, void *buf2, unsigned int cnt);
```

Description

memicmp compares the first *cnt* bytes of *buf1* and *buf2* without regard to the case of letters in the two buffers. The function converts all uppercase characters into lowercase and then performs the comparison.

Returns

The return value of memicmp indicates the result as follows: compact break=fit.

Value	**Meaning**
Less than 0	*buf1* less than *buf2*
0	*buf1* identical to *buf2*
Greater than 0	*buf1* greater than *buf2*.

Example Code

This example copies two strings that each contain a substring of 29 characters that are the same except for case. The example then compares the first 29 bytes without regard to case.

```
#include <stdio.h>
#include <string.h>

char first[100],second[100];

int main(void)
{
   int result;

   strcpy(first, "Those Who Will Not Learn From History");
   strcpy(second, "THOSE WHO WILL NOT LEARN FROM their mistakes");
   printf("Comparing the first 29 characters of two strings.\n");
   result = memicmp(first, second, 29);
   printf("The first 29 characters of String 1 are ");
   if (result < 0)
      printf("less than String 2.\n");
   else
      if (0 == result)
         printf("equal to String 2.\n");
      else
         printf("greater than String 2.\n");
   return 0;

   /*****************************************************************************
      The output should be:
```

```
      Comparing the first 29 characters of two strings.
      The first 29 characters of String 1 are equal to String 2
   ****************************************************************************/
}
```

Related Information

- memchr
- memcmp
- memcpy
- memmove
- memset
- strcmp

----

memmove - Copy Bytes

memmove - Copy Bytes

Syntax

```
#include <string.h>  /* also in <memory.h> */
void *memmove(void *dest, const void *src, size_t count);
```

Description

`memmove` copies *count* bytes of *src* to *dest*. `memmove` allows copying between objects that may overlap as if *src* is first copied into a temporary array.

Returns

`memmove` returns a pointer to *dest*.

Example Code

This example copies the word `shiny` from position `target + 2` to position `target + 8`.

```
#include <string.h>
#include <stdio.h>

#define  SIZE          21
char target[SIZE] = "a shiny white sphere";

int main(void)
{
   char *p = target+8;                    /* p points at the starting character
                                             of the word we want to replace   */
   char *source = target+2;                                /* start of "shiny" */

   printf("Before memmove, target is \"%s\"\n", target);
   memmove(p, source, 5);
   printf("After memmove, target becomes \"%s\"\n", target);
   return 0;

   /****************************************************************************
      The output should be:

      Before memmove, target is "a shiny white sphere"
      After memmove, target becomes "a shiny shiny sphere"
   ****************************************************************************/
}
```

Related Information

- memchr
- memcmp
- memcpy
- memset
- strcpy

----

memset - Set Bytes to Value

memset - Set Bytes to Value

Syntax

```
#include <string.h>  /* also in <memory.h> */
void *memset(void *dest, int c, size_t count);
```

Description

`memset` sets the first *count* bytes of *dest* to the value *c*. The value of *c* is converted to an unsigned *char*.

Returns

`memset` returns a pointer to *dest*.

Example Code

This example sets 10 bytes of the buffer to A and the next 10 bytes to B.

```
#include <string.h>
#include <stdio.h>

#define  BUF_SIZE      20

int main(void)
{
   char buffer[BUF_SIZE+1];
   char *string;

   memset(buffer, 0, sizeof(buffer));
   string = memset(buffer, 'A', 10);
   printf("\nBuffer contents: %s\n", string);
   memset(buffer+10, 'B', 10);
   printf("\nBuffer contents: %s\n", buffer);
   return 0;

   /****************************************************************************
      The output should be:

      Buffer contents: AAAAAAAAAA
      Buffer contents: AAAAAAAAAABBBBBBBBBB
   ****************************************************************************/
}
```

Related Information

- memchr
- memcmp
- memcpy
- memicmp

- memmove
- strnset - strset

_mheap - Query Memory Heap for Allocated Object

_mheap - Query Memory Heap for Allocated Object

Syntax

```
#include <umalloc.h>
Heap_t _mheap(void *ptr);
```

Description

_mheap determines from which heap the object specified by *ptr* was allocated. The *ptr* must be a valid pointer that was returned from a run-time allocation function (_ucalloc, malloc, realloc, and so on). If the pointer is not valid, the results of _mheap are undefined.

For more information about creating and using heaps, see the chapter on Managing Memory in the *VisualAge C++ Programming Guide* .

Returns

_mheap returns the handle of the heap from which the object was allocated. If the object was allocated from the run-time heap, _mheap returns _RUNTIME_HEAP. If the object passed to _mheap is NULL, _mheap returns NULL. If the object is not valid, _mheap either returns NULL (depending on how closely the storage pointed to resembles a valid object), or an exception occurs.

Example Code

This example allocates a block of memory from the heap, then uses _mheap to determine which heap the block came from.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   char  *ptr;

   if (NULL == (ptr = malloc(10))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   printf("Handle of heap used is 0x%x\n", _mheap(ptr));
   return 0;

   /****************************************************************************
      The output should be similar to :

      Handle of heap used is 0x70000
   ****************************************************************************/
}
```

Related Information

- "Managing Memory" in the *VisualAge C++ Programming Guide*
- _msize
- _ucreate
- _ustats

----

min - Return Lesser of Two Values

min - Return Lesser of Two Values

Syntax

```
#include <stdlib.h>
type min(type a, type b);
```

Description

`min` compares two values and determines the smaller of the two. The data *type* can be any arithmetic data type, signed or unsigned. The *type* must be the same for both arguments to min.

Note: Because `min` is a macro, if the evaluation of the arguments contains side effects (post-increment operators, for example), the results of both the side effects and the macro will be undefined.

Returns

`min` returns the smaller of the two values.

Example Code

This example prints the smaller of the two values, `a` and `b`.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int a = 10;
   int b = 21;

   printf("The smaller of %d and %d is %d\n", a, b, min(a, b));
   return 0;

   /*****************************************************************************
      The output should be:

      The smaller of 10 and 21 is 10
   *****************************************************************************/
}
```

Related Information

- max

----

mkdir - Create New Directory

mkdir - Create New Directory

Syntax

```
#include <direct.h>
int mkdir(char *pathname);
```

Description

mkdir creates a new directory with the specified *pathname* . Because only one directory can be created at a time, only the last component of *pathname* can name a new directory.

Returns

mkdir returns the value `0` if the directory was created. A return value of -1 indicates an error, and errno is set to one of the following values: compact break=fit.

Value	Meaning
EACCESS	The directory was not created; the given name is the name of an existing file, directory, or device.
ENOENT	The *pathname* was not found.

Example Code

This example creates two new directories: one at the root on drive `C:`, and one in the `tmp` subdirectory of the current working directory.

```
#include <stdio.h>
#include <direct.h>
#include <string.h>

int main(void)
{
   char *dir1,*dir2;

 /*  Create the directory "aleng" in the root directory of the C: drive.      */

   dir1 = "c:\\aleng";
   if (0 == (mkdir(dir1)))
      printf("%s directory was created.\n", dir1);
   else
      printf("%s directory was not created.\n", dir1);

 /*  Create the subdirectory "simon" in the current directory.                */

   dir2 = "simon";
   if (0 == (mkdir(dir2)))
      printf("%s directory was created.\n", dir2);
   else
      printf("%s directory was not created.\n", dir2);

 /*  Remove the directory "aleng" from the root directory of the C: drive.    */

   printf("Removing directory 'aleng' from the root directory.\n");
   if (rmdir(dir1))
      perror(NULL);
   else
      printf("%s directory was removed.\n", dir1);

 /*  Remove the subdirectory "simon" from the current directory.              */

   printf("Removing subdirectory 'simon' from the current directory.\n");
   if (rmdir(dir2))
      perror(NULL);
   else
      printf("%s directory was removed.\n", dir2);
   return 0;

   /****************************************************************************
      The output should be:

      c:\aleng directory was created.
      simon directory was created.
      Removing directory 'aleng' from the root directory.
      c:\aleng directory was removed.
      Removing subdirectory 'simon' from the current directory.
      simon directory was removed.
   ****************************************************************************/
}
```

Related Information

- chdir
- _getcwd
- _getdcwd
- rmdir

----

mktime - Convert Local Time

mktime - Convert Local Time

Syntax

```
#include <time.h>
time_t mktime(struct tm *time);
```

Description

`mktime` converts local time, stored as a `tm` structure pointed to by *time* , into a `time_t` structure suitable for use with other time functions. The values of some structure elements pointed to by *time* are not restricted to the ranges shown for gmtime.

The values of `tm_wday` and `tm_yday` passed to `mktime` are ignored and are assigned their correct values on return.

Note: The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Returns

`mktime` returns the calendar time having type `time_t`. The value `(time_t)(-1)` is returned if the calendar time cannot be represented.

Example Code

This example prints the day of the week that is 40 days and 16 hours from the current date.

```
#include <stdio.h>
#include <time.h>

char *wday[] =  { "Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday" } ;

int main(void)
{
   time_t t1,t3;
   struct tm *t2;

   t1 = time(NULL);
   t2 = localtime(&t1);
   t2->tm_mday += 40;
   t2->tm_hour += 16;
   t3 = mktime(t2);
   printf("40 days and 16 hours from now, it will be a %9s \n", wday[t2->tm_wday
      ]);
   return 0;
   /****************************************************************************
      The output should be similar to:

      40 days and 16 hours from now, it will be a Sunday
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- localtime
- time

----

modf - Separate Floating-Point Value

modf - Separate Floating-Point Value

Syntax

```
#include <math.h>
double modf(double x, double *intptr);
```

Description

`modf` breaks down the floating-point value *x* into fractional and integral parts. The signed fractional portion of *x* is returned. The integer portion is stored as a double value pointed to by *intptr*. Both the fractional and integral parts are given the same sign as *x*.

Returns

`modf` returns the signed fractional portion of *x*.

Example Code

This example breaks the floating-point number -14.876 into its fractional and integral components.

```
#include <math.h>

int main(void)
{
   double x,y,d;

   x = -14.876;
   y = modf(x, &d);
   printf("x = %lf\n", x);
   printf("Integral part = %lf\n", d);
   printf("Fractional part = %lf\n", y);
   return 0;

   /****************************************************************************
      The output should be:

      x = -14.876000
      Integral part = -14.000000
      Fractional part = -0.876000
   ****************************************************************************/
}
```

Related Information

- fmod
- frexp
- ldexp

_msize - Return Number of Bytes Allocated

_msize - Return Number of Bytes Allocated

Syntax

```
#include <stdlib.h>  /* also in <malloc.h> */
size_t _msize(void *ptr)
```

Description

_msize determines the number of bytes that were allocated to the pointer argument *ptr*. The *ptr* must have been returned from one of the run-time memory allocation functions (_ucalloc, malloc, _trealloc, and so on).

You cannot pass the argument of an object that has been freed.

Returns

_msize returns the number of bytes allocated. If the argument is not a valid pointer returned from a memory allocation function, the return value is undefined. If NULL is passed, _msize returns 0.

Example Code

This example displays the size of an allocated object from malloc.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *ptr;

   if (NULL == (ptr = malloc(10))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 5);
   printf("The size of the allocated object is %u.\n",_msize(ptr));
   return 0;

   /****************************************************************************
      The output should be similar to :

      The size of the allocated object is 10.
   ****************************************************************************/
}
```

Related Information

- calloc
- malloc
- realloc

----

nl_langinfo - Retrieve Locale Information

nl_langinfo - Retrieve Locale Information

Syntax

```
#include <langinfo.h>
char *nl_langinfo(nl_item item);
```

Description

nl_langinfo retrieves from the current locale the string that describes the requested information specified by *item* .

The constant names and values for *item* are defined in `<langinfo.h>` which includes `<ulsitem.h>`, the header file that actually contains the constant names and values.

Returns

nl_langinfo returns a pointer to a null-terminated string containing information about the active language or cultural area. The active language or cultural area is determined by the most recent setlocale call. Subsequent calls to the function may modify the array that the return value points to. Your own code cannot modify the array.

If *item* is not valid, nl_langinfo returns a pointer to an empty string.

Example Code

This example uses nl_langinfo to retrieve the current codeset name.

```
#include <langinfo.h>
#include <stdio.h>

int main(void)
{
   printf("Current codeset is %s\n", nl_langinfo(CODESET));
   return 0;

   /*****************************************************************************
      The output should be similar to :

      Current codeset is IBM-850
   *****************************************************************************/
}
```

Related Information

- localeconv
- setlocale

_onexit - Record Termination Function

_onexit - Record Termination Function

Syntax

```
#include <stdlib.h>
```

```
onexit_t _onexit(onexit_t func);
```

Description

_onexit records the address of a function *func* to call when the program ends normally. Successive calls to _onexit create a stack of functions that run in a last-in-first-out order. The functions passed to _onexit cannot take parameters.

You can record up to 32 termination functions with calls to _onexit and atexit. If you exceed 32 functions, _onexit returns the value `NULL`.

Note: For portability, use the ANSI/ISO standard atexit function, which is equivalent to _onexit.

Returns

If successful, _onexit returns a pointer to the function; otherwise, it returns a `NULL` value.

Example Code

This example specifies and defines four distinct functions that run consecutively at the completion of `main`.

```
#include <stdio.h>
#include <stdlib.h>

int fn1(void)
{
   printf("next.\n");
}

int fn2(void)
{
   printf("run ");
}

int fn3(void)
{
   printf("is ");
}

int fn4(void)
{
   printf("This ");
}

int main(void)
{
   _onexit(fn1);
   _onexit(fn2);
   _onexit(fn3);
   _onexit(fn4);
   printf("This is run first.\n");
   return 0;

   /****************************************************************************
      The output should be:

      This is run first.
      This is run next.
   ****************************************************************************/
}
```

Related Information

- abort
- atexit
- exit
- _exit

----

open - Open File

open - Open File

Syntax

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
int open(char *pathname, int oflag, int pmode);
```

Description

open opens the file specified by *pathname* and prepares the file for subsequent reading or writing as defined by *oflag* . open can also prepare the file for reading and writing.

The *oflag* is an integer expression formed by combining one or more of the following constants, defined in `<fcntl.h>`. To specify more than one constant, join the constants with the bitwise OR operator (|); for example, O_CREAT | O_TEXT.

Oflag	**Meaning**
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_CREAT	Create and open a new file. This flag has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. This flag applies only when used with O_CREAT.
O_RDONLY	Open the file for reading only. If this flag is given, neither O_RDWR nor O_WRONLY can be given.
O_RDWR	Open the file for reading and writing. If this flag is given, neither O_RDONLY nor O_WRONLY can be given.
O_TRUNC	Open and truncate an existing file to `0` length. The file must have write permission. The contents of the file are destroyed, and O_TRUNC cannot be specified with O_RDONLY.
O_WRONLY	Open the file for writing only. If this flag is given, neither O_RDONLY nor O_RDWR can be given.
O_BINARY	Open the file in binary (untranslated) mode.
O_TEXT	Open the file in text (translated) mode.

If neither O_BINARY or O_TEXT is specified, the default will be O_TEXT; it is an error to specify both O_BINARY and O_TEXT. You must specify one of the access mode flags, O_RDONLY, O_WRONLY, or O_RDWR. There is no default.

Warning: Use O_TRUNC with care; it destroys the complete contents of an existing file.

For more details on text and binary modes and their differences, see "Stream Processing" in the *VisualAge C++ Programming Guide* .

The *pmode* argument is an integer expression containing one or both of the constants S_IWRITE and S_IREAD, defined in `<sys\stat.h>`. The *pmode* is required only when O_CREAT is specified. If the file exists, *pmode* is ignored. Otherwise, *pmode* specifies the permission settings for the file. These are set when the new file is closed for the first time. The meaning of the *pmode* argument is as follows: compact break=fit.

Value	**Meaning**
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If write permission is not given, the file is read-only. Under the OS/2 operating system, all files are readable; you cannot give write-only permission. The modes S_IWRITE and S_IREAD | S_IWRITE are equivalent.

open applies the current file permission mask to *pmode* before setting the permissions. (See umask.)

Note: In earlier releases of C Set ++, open began with an underscore (`_open`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_open` to open for you.

Returns

open returns a file handle for the opened file. A return value of `-1` indicates an error, and `errno` is set to one of the following values: compact break=fit.

Value	Meaning
EACCESS	The given *pathname* is a directory; or the file is read-only but an open for writing was attempted; or a sharing violation occurred.
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EMFILE	No more file handles are available.
EINVAL	An incorrect argument was passed.
ENOENT	The file or *pathname* were not found.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example opens the file `edopen.dat` by creating it as a new file, truncating it if it exists, and opening it so it can be read and written to. The open command issued also grants permission to read from and write to the file.

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>

int main(void)
{
   int fh;

   if (-1 == (fh = open("edopen.dat", O_CREAT|O_TRUNC|O_RDWR,
                         S_IREAD|S_IWRITE))) {
      perror("Unable to open edopen.dat");
      return EXIT_FAILURE;
   }
   printf("File was successfully opened.\n");
   if (-1 == close(fh)) {
      perror("close error");
      return EXIT_FAILURE;
   }
   return 0;

   /****************************************************************************
      The output should be:

      File was successfully opened.
   ****************************************************************************/
}
```

Related Information

- close
- creat
- fdopen
- fopen
- _sopen
- umask

----

perror - Print Error Message

perror - Print Error Message

Syntax

```
#include <stdio.h>
void perror(const char *string);
```

Description

`perror` prints an error message to stderr. If *string* is not `NULL` and does not point to a null character, the string pointed to by *string* is printed to the standard error stream, followed by a colon and a space. The message associated with the value in `errno` is then printed followed by a new-line character.

To produce accurate results, you should ensure that `perror` is called immediately after a library function returns with an error; otherwise, subsequent calls may alter the `errno` value.

Returns

There is no return value.

Example Code

This example tries to open a stream. If `fopen` fails, the example prints a message and ends the program.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   FILE *fh;

   if (NULL == (fh = fopen("myfile.mjq", "r"))) {
      perror("Could not open data file");
      abort();
   }
   return 0;

   /****************************************************************************
      The output should be:

      Could not open data file: The file cannot be found.
   ****************************************************************************/
}
```

Related Information

- clearerr
- ferror
- strerror
- _strerror

----

pow - Compute Power

pow - Compute Power

Syntax

```
#include <math.h>
double pow(double x, double y);
```

Description

`pow` calculates the value of *x* to the power of *y*.

Returns

If *y* is `0`, `pow` returns the value 1. If *x* is `0` and *y* is negative, `pow` sets `errno` to `EDOM` and returns `0`. If both *x* and *y* are `0`, or if *x* is negative and *y* is not an integer, `pow` sets `errno` to `EDOM`, and returns `0`.

If an overflow results, `pow` sets `errno` to `ERANGE` and returns `+HUGE_VAL` for a large result or `-HUGE_VAL` for a small result.

Example Code

This example calculates the value of 2^3.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double x,y,z;

   x = 2.0;
   y = 3.0;
   z = pow(x, y);
   printf("%lf to the power of %lf is %lf\n", x, y, z);
   return 0;

   /****************************************************************************
      The output should be:

      2.000000 to the power of 3.000000 is 8.000000
   ****************************************************************************/
}
```

Related Information

- exp
- log
- log10
- sqrt

----

printf - Print Formatted Characters

printf - Print Formatted Characters

Syntax

```
#include <stdio.h>
int printf(const char *format-string, argument-list);
```

Description

printf formats and prints a series of characters and values to the standard output stream stdout. The *format-string* consists of ordinary characters, escape sequences, and format specifications. The ordinary characters are copied in order of their appearance to stdout. Format specifications, beginning with a percent sign (%), determine the output format for any *argument-list* following the *format-string* .

The *format-string* is read left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* after the *format-string* to be converted and output, and so on through the end of the *format-string* . If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. A format specification has the following form:

```
>>  %                                                         type  ><
        n$     flags     width     .  precision     h
                                                        l
                                                        L
```

Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

The `%n$` syntax should always be used when format strings are stored in translatable text such as message catalog files. This is necessary because sentence structures and order of arguments could vary in different languages. For example, in the statement: `printf ( format, month, day, year )`

- In US English, `format` should be `%1$s %2$d, %3$d` which results in July 14, 1995.
- In German, `format` should be `%2$d. %1$s, %3$d` which results in 14. July, 1995.

The following optional fields control other aspects of the formatting:

Field	**Description**	
flags	Justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for `wchar_t` precision unit.	
width	Minimum number of bytes output.	
precision	Maximum number of bytes printed for all or part of the output field, or minimum number of digits printed for integer values.	
`h, l, L`	Size of argument expected: compact break=fit.	
	h	A prefix with the integer types `d`, `i`, `o`, `u`, `x`, `X`, and `n` that specifies that the argument is `short int` or `unsigned short int`.
	l	A prefix with `d`, `i`, `o`, `u`, `x`, `X`, and `n` types that specifies that the argument is a `long int` or `unsigned long int`.
	L	A prefix with `e`, `E`, `f`, `g`, or `G` types that specifies that the argument is `long double`.

Each field of the format specification is discussed in detail below. To print a percent sign character, use %%.

In extended mode, `printf` also converts floating-point values of NaN and infinity to the strings `"NAN"` or `"nan"` and `"INFINITY"` or `"infinity"`. The case and sign of the string is determined by the format specifiers. See Infinity and NaN Support for more information on infinity and NaN values.

The *type* characters and their meanings are given in the following table:

Character	Argument	Output Format
d, i	Integer	Signed decimal integer.
o	Integer	Unsigned octal integer.
u	Integer	Unsigned decimal integer.
x	Integer	Unsigned hexadecimal integer, using abcdef.
X	Integer	Unsigned hexadecimal integer, using ABCDEF.

f	Double	Signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point is equal to the requested precision. NaN and infinity values are printed in lowercase ("nan" and "infinity").
F	Double	In extended mode,identical to the "f" format except that NaN and infinity values are printed in uppercase ("NAN" and "INFINITY"). In modes other than extended, "F" is treated like any other character not included in this table.
e	Double	Signed value having the form [-]d.dddd"e"[sign] ddd, where d is a single-decimal digit, dddd is one or more decimal digits, ddd is 2 or 3 decimal digits, and sign is + or -.
E	Double	Identical to the "e" format except that "E" introduces the exponent instead of "e".
g	Double	Signed value printed in "f" or "e" format. The "e" format is used only when the exponent of the value is less than -4 or greater than precision. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Double	Identical to the "g" format except that "G" introduces the exponent (where appropriate) instead of "g".
c	Character	Single character. The "int" argument is converted to an unsigned character.
lc, C	Wide character	Multibyte character (converted as if by a call to wctomb) pointer to an array of "char".
s	String	Characters printed up to the first null character (\"0") or until precision is reached. If you specify a null string, "(NULL)" is printed.
ls, S	Wide-character string	Multibyte characters, printed up to the first "wchar_t" null character ("L\0") is encountered in the wide-character string, or until the specified precision is reached. Conversion takes place as if by a call to wcstombs. The displayed result does not include the terminating null character. If you do not specify the precision, you must end the wide-character string with a null character. A partial multibyte char-

		acter cannot be written. If you specify a null string, "(NULL)" is printed.
n	Pointer to integer	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters.
%	`%'	Use to print a "%" symbol.

The *flag* characters and their meanings are as follows (notice that more than one *flag* can appear in a format specification):

Flag	Meaning	Default
-	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or -) if the output value is of a signed type.	Sign appears only for negative signed values (-).
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The "+" flag overrides the blank flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the "o", "x", or "X" formats, the "#" flag prefixes any nonzero output value with "0", "0"x, or "0"X, respectively.	No prefix.
#	When used with the "f", "F", "e", or "E" formats, the "#" flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
#	When used with the "g" or "G" formats, the "#" flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it; trailing zeros are truncated.
#	When used with the "ls" format, the "#" flag causes precision to be measured in "wchar_t" characters.	Precision indicates the maximum number of bytes to be output.
"0"	When used with the "d", "i", "o", "u", "x", "X", "e", "E", "f", "F"" g", or "G" formats, the "0" flag causes leading "0"'s to pad the output to the field width. The "0" flag is	Space padding.

	ignored if precision is specified for an integer or if the `"-"' flag is specified.	
'	Formats with the thousands grouping character of the appropriate locale for the decimal conversions (%i, %d, %u, %f, %g, or %G).	No grouping character.

The `#` flag should not be used with `c`, `lc`, `d`, `i`, `u`, `s`, or `p` types.

Width is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width* , blanks are added on the left or the right (depending on whether the `-` flag is specified) until the minimum width is reached.

Width never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width* , or *width* is not given, all characters of the value are printed (subject to the *precision* specification).

For the `ls` type, *width* is specified in bytes. If the number of bytes in the output value is less than the specified width, single-byte blanks are added on the left or the right (depending on whether the `-` flag is specified) until the minimum width is reached.

The *width* specification can be an asterisk (* or *n$), in which case an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list.

Precision is a nonnegative decimal integer preceded by a period, which specifies the number of characters to be printed or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value or rounding of a floating-point value.

The *precision* specification can be an asterisk (* or *n$), in which case an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type* , as shown in the following table:

Type	Meaning	Default
i d u o x X	Precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than precision, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds precision.	If precision is "0" or omitted entirely, or if the period (.) appears without a number following it, the precision is set to 1.
f F e E	Precision specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.	Default precision is six. If precision is "0" or the period appears without a number following it, no decimal point is printed.
g G	Precision specifies the maximum number of significant digits printed.	All significant digits are printed.
c	No effect.	The character is printed.
lc, C	No effect.	The "wchar_t" character is printed.
s	Precision specifies the maximum number of characters to be printed. Characters in excess	Characters are printed until a null character is encountered.

	of precision are not printed.	
ls, S	Precision specifies the maximum number of bytes to be printed. Bytes in excess of precision are not printed; however, multibyte integrity is always preserved.	"wchar_t" characters are printed until a null character is encountered.

Returns

The `printf` function returns the number of bytes printed.

Example Code

This example prints data in a variety of formats.

```
#include <stdio.h>

int main(void)
{
   char ch = 'h',*string = "computer";
   int count = 234,hex = 0x10,oct = 010,dec = 10;
   double fp = 251.7366;

   printf("%d    %+d    %06d     %X    %x     %o\n\n", count, count, count, count
      , count, count);
   printf("1234567890123%n4567890123456789\n\n", &count);
   printf("Value of count should be 13; count = %d\n\n", count);
   printf("%10c%5c\n\n", ch, ch);
   printf("%25s\n%25.4s\n\n", string, string);
   printf("%f    %.2f    %e    %E\n\n", fp, fp, fp, fp);
   printf("%i    %i     %i\n\n", hex, oct, dec);
   return 0;

   /****************************************************************************
      The output should be:

      234   +234    000234     EA    ea     352

      12345678901234567890123456789

      Value of count should be 13; count = 13

               h    h

                       computer
                           comp

      251.736600    251.74    2.517366e+02    2.517366E+02

      16    8     10
   ****************************************************************************/
}
```

Related Information

- _cprintf
- fprintf
- fscanf
- scanf
- sprintf
- sscanf
- vfprintf
- vprintf
- vsprintf

----

putc - putchar - Write a Byte

putc - putchar - Write a Byte

Syntax

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

Description

`putc` converts *c* to `unsigned char` and then writes *c* to the output *stream* at the current position. `putchar` is equivalent to `putc(`*c*`, stdout)`.

`putc` is equivalent to fputc except that, if it is implemented as a macro, `putc` can evaluate *stream* more than once. Therefore, the *stream* argument to `putc` should not be an expression with side effects.

Returns

`putc` and `putchar` return the value written. A return value of `EOF` indicates an error.

Example Code

This example writes the contents of a buffer to a data stream. In this example, the body of the `for` statement is null because the example carries out the writing operation in the test expression.

```
#include <stdio.h>

#define  LENGTH        80

int main(void)
{
   FILE *stream = stdout;
   int i,ch;
   char buffer[LENGTH+1] = "Hello world";

    /* This could be replaced by using the fwrite routine                     */

   for (i = 0; (i < sizeof(buffer)) && ((ch = putc(buffer[i], stream)) != EOF);
      ++i)
      ;
   return 0;

   /****************************************************************************
      The output should be:

      Hello world
   ****************************************************************************/
}
```

Related Information

- fputc
- fwrite
- getc - getchar
- _putch
- puts
- write

_putch - Write Character to Screen

_putch - Write Character to Screen

Syntax

```
#include <conio.h>
int _putch(int c);
```

Description

_putch writes the character *c* directly to the screen.

Returns

If successful, _putch returns *c*. If an error occurs, _putch returns `EOF`.

Example Code

This example defines a function `gchar` that is similar to _getche using the _putch and _getch functions:

```
#include <conio.h>

int gchar(void)
{
   int ch;

   ch = _getch();
   _putch(ch);
   return (ch);
}
```

Related Information

- _cputs
- _cprintf
- fputc
- _getch - _getche
- putc - putchar
- puts
- write

----

putenv - Modify Environment Variables

putenv - Modify Environment Variables

Syntax

```
#include <stdlib.h>
int putenv(char *envstring);
```

Description

putenv adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process runs (for example, the default search path for libraries to be linked with a program).

The *envstring* argument must be a pointer to a string with the form:

```
   varname=string
```

where *varname* is the name of the environment variable to be added or modified and *string* is the value of the variable. See the **Notes** below.

If *varname* is already part of the environment, *string* replaces its current value; if not, the new *varname* is added to the environment with the value *string*. To set a variable to an empty value, specify an empty *string*. A variable can be removed from the environment by specifying *varname* only, for example:

```
   putenv("PATH");
```

Do not free the *envstring* pointer while the entry it points to is in use, or the environment variable will point into freed space. A similar problem can occur if you pass a pointer to a local variable to putenv and then exit from the function in which the variable is declared. Once you have added the *envstring* with putenv, any change to the entry it points to changes the environment used by your program.

The environment manipulated by putenv is local to the process currently running. You cannot enter new items in your command-level environment using putenv. When the program ends, the environment reverts to the parent process environment. This environment is passed on to some child processes created by the _spawn, exec, or system functions, and they get any new environment variables added using putenv.

`DosScanEnv` will not reflect any changes made using putenv, but `getenv` will reflect the changes.

Note:

- putenv can change the value of _environ, thus invalidating the `envp` argument to the `main` function.
- You cannot use %*envvar*%, where *envvar* is any OS/2 environment variable, with putenv to concatenate new *envstring* and old *envstring*.
- In earlier releases of C Set ++, putenv began with an underscore (`_putenv`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_putenv` to putenv for you.

Returns

putenv returns `0` if it is successful. A return value of `-1` indicates an error.

Example Code

This example tries to change the environment variable PATH, and then uses getenv to get the current path. If the call to putenv fails, the example writes an error message.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *pathvar;

   if (-1 == putenv("PATH=a:\\bin;b:\\andy")) {
      printf("putenv failed - out of memory\n");
      return EXIT_FAILURE;
   }

   /* getting and printing the current environment path                       */

   pathvar = getenv("PATH");
   printf("The current path is: %s\n", pathvar);
   return 0;

   /****************************************************************************
      The output should be:

      The current path is: a:\bin;b:\andy
   ****************************************************************************/
}
```

Related Information

- execl - _execvpe
- getenv
- _spawnl - _spawnvpe
- system
- "envp Parameter to main" in the *VisualAge C++ Programming Guide*

----

puts - Write a String

puts - Write a String

Syntax

```
#include <stdio.h>
int puts(const char *string);
```

Description

`puts` writes the given *string* to the standard output stream stdout; it also appends a new-line character to the output. The terminating null character is not written.

Returns

`puts` returns `EOF` if an error occurs. A nonnegative return value indicates that no error has occurred.

Example Code

This example writes `Hello World` to stdout.

```
#include <stdio.h>

int main(void)
{
   if (EOF == puts("Hello World"))
      printf("Error in puts\n");
   return 0;

   /****************************************************************************
      The output should be:

      Hello World
   ****************************************************************************/
}
```

Related Information

- _cputs
- fputs
- gets
- putc - putchar

----

putwc - Write Wide Character

putwc - Write Wide Character

Syntax

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t wc, FILE *stream);
```

Description

putwc converts the wide character *wc* to a multibyte character, and writes it to the *stream* at the current position. It also advances the file position indicator for the stream appropriately.

putwc function is equivalent to fputwc except that, if it is implemented as a macro, putwc can evaluate *stream* more than once. Therefore, the *stream* argument to putwc should not be an expression with side effects.

The behavior of putwc is affected by the LC_CTYPE category of the current locale.

After calling putwc, flush the buffer or reposition the stream pointer before calling a read function for the stream, unless EOF has been reached. After a read operation on the stream, flush the buffer or reposition the stream pointer before calling putwc.

Returns

putwc returns the wide character written. If a write error occurs, putwc sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, putwc sets errno to EILSEQ and returns WEOF.

Example Code

The following example uses putwc to convert the wide characters in `wcs` to multibyte characters and write them to the file `putwc.out`.

```
#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"A character string.";
   int     i;

   if (NULL == (stream = fopen("putwc.out", "w"))) {
      printf("Unable to open: \"putwc.out\".\n");
      exit(EXIT_FAILURE);
   }

   for (i = 0; wcs[i] != L'\0'; i++) {
      errno = 0;
      if (WEOF == putwc(wcs[i], stream)) {
         printf("Unable to putwc() the wide character.\n"
                "wcs[%d] = 0x%lx\n", i, wcs[i]);
         if (EILSEQ == errno)
            printf("An invalid wide character was encountered.\n");
         exit(EXIT_FAILURE);
      }
   }
   fclose(stream);
   return 0;

   /****************************************************************************
      The output file putwc.out should contain :

      A character string.
   ****************************************************************************/
}
```

Related Information

- fputc
- fputwc
- getwc
- putc - putchar
- _putch
- putwchar

----

putwchar - Write Wide Character to stdout

putwchar - Write Wide Character to stdout

Syntax

```
#include <wchar.h>
wint_t putwchar(wchar_t wc);
```

Description

putwchar converts the wide character *wc* to a multibyte character and writes it to stdout. A call to putwchar is equivalent to `putwc(wc, stdout)`.

The behavior of putwchar is affected by the LC_CTYPE category of the current locale.

After calling putwchar, flush the buffer or reposition the stream pointer before calling a read function for the stream, unless EOF has been reached. After a read operation on the stream, flush the buffer or reposition the stream pointer before calling putwchar.

Returns

putwchar returns the wide character written. If a write error occurs, putwchar sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, putwchar sets errno to EILSEQ and returns WEOF.

Example Code

This example uses putwchar to write the string in `wcs`.

```
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
   wchar_t *wcs = L"A character string.";
   int     i;

   for (i = 0; wcs[i] != L'\0'; i++) {
      errno = 0;
      if (WEOF == putwchar(wcs[i])) {
         printf("Unable to putwchar() the wide character.\n");
         printf("wcs[%d] = 0x%lx\n", i, wcs[i]);
         if (EILSEQ == errno)
            printf("An invalid wide character was encountered.\n");
         exit(EXIT_FAILURE);
      }
   }
   return 0;

   /***************************************************************************
      The output should be similar to :
```

```
      A character string.
   ***************************************************************************/
}
```

Related Information

- fputc
- fputwc
- getwchar
- putc - putchar
- _putch

----

qsort - Sort Array

qsort - Sort Array

Syntax

```
#include <stdlib.h>
void qsort(void *base, size_t num, size_t width,
           int(*compare)(const void *key, const void *element));
```

Description

`qsort` sorts an array of *num* elements, each of *width* bytes in size. The *base* pointer is a pointer to the array to be sorted. `qsort` overwrites this array with the sorted elements.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. qsort calls this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values: compact break=fit.

Value	Meaning
Less than `0`	*key* less than *element*
`0`	*key* equal to *element*
Greater than `0`	*key* greater than *element*

The sorted array elements are stored in ascending order, as defined by your *compare* function. You can sort in reverse order by reversing the sense of "greater than" and "less than" in *compare*. The order of the elements is unspecified when two elements compare equally.

Returns

There is no return value.

Example Code

This example sorts the arguments (`argv`) in ascending lexical sequence, using the comparison function `compare()` supplied in the example.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* -------------------------------------------------------------              */
/* compare() routine called internally by qsort()                             */
/* -------------------------------------------------------------              */
```

```
int compare(const void *arg1,const void *arg2)
{
   return (strcmp(*(char **)arg1, *(char **)arg2));
}

int main(int argc,char *argv[])
{
   int i;
   argv++;
   argc--;

   qsort((char *)argv, argc, sizeof(char *), compare);
   for (i = 0; i < argc; ++i)
      printf("%s\n", argv[i]);
   return 0;

   /****************************************************************************
      Assuming command line of: qsort kent theresa andrea laura brenden
      Output should be:

        andrea
        brenden
        kent
        laura
        theresa
   ****************************************************************************/
}
```

Related Information

- bsearch
- lfind - lsearch

----

querylocaleenv - Query Locale Environment Variables

querylocaleenv - Query Locale Environment Variables

Syntax

```
#include <locale.h>
char * querylocaleenv (void)
```

Description

querylocaleenv searches the process environment to resolve the locale object category values according to the following priority:

- If the LC_ALL environment variable is defined and is not NULL, the value of the LC_ALL environment variable is used.

- If the LC_* environment variable or variables are defined and not NULL, the value of the environment variable is used to initialize the category that corresponds to the environment variable. The LC_ variables include LC_CTYPE, LC_COLLATE, LC_TIME, LC_MONETARY, LC_NUMERIC, and LC_MESSAGES.

- If the LANG environment variable is defined and is not NULL, the value of the LANG environment variable is used.

The LANG environment variable has the lowest priority of all the environment variables used by querylocaleenv to initialize locale categories.

The following actions occur in the absence of the environment variables:

- In the absence of the LC_ALL environment variable, the value of the LANG environment variable is used to determine the value of the category for which the environment variable (LC_CTYPE, LC_TIME, LC_MONETARY, LC_NUMERIC, or LC_MESSAGES) that corresponds to that category is not defined or is NULL.

- In the absence of the LC_ALL environment variable, the LC_* environment variables, and the LANG environment variable, querylocaleenv returns a string containing UNIV as the result.

Returns

querylocaleenv returns a pointer to a NULL terminated *char* string that represents the result of the search of the locale environment variables. The pointer points at memory that has been allocated by calling malloc. You can choose to return this memory by calling the free function.

The format of the returned category string is a NULL terminated set of 7-bit ASCII encoded characters. The returned string contains the values of the following environment variables listed in the order that they are returned in the string:

LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
LC_MESSAGES

The value of the environment variables is a string with the following format made up of the name of the locale that is currently selected:

```
language_territory
```

where *language* is the ISO 639 two-letter name and *territory* is the ISO 3166 two-letter territory (country) name. For example, `fr_FR` represents French in France.

Example Code

```
#include <unidef.h>
#include <locale.h>

/* ------------------------------------------------------------             */
/* Pointer to contents of locale environment variables                      */
/* ------------------------------------------------------------             */

char           *pEnv;
int            rc;
LocaleObject   locale_object;

/* ------------------------------------------------------------             */
/* Test pointer returned from querylocaleenv call                           */
/* ------------------------------------------------------------             */

if ((pEnv=querylocalenv)==NULL)
{

/* ------------------------------------------------------------             */
/* Error indicating out-of-memory                                           */
/* ------------------------------------------------------------             */

} else
{
    rc=UniCreateLocaleObject(UNI_MBS_STRING_POINTER,pEnv,
            &locale_object);

/* ------------------------------------------------------------             */
/* locale_object can be used in subsequent calls to ULS APIs             */
/* ------------------------------------------------------------             */

    free(pEnv);
}
```

Related Information

- bsearch
- lfind - lsearch

----

raise - Send Signal

raise - Send Signal

Syntax

```
#include <signal.h>
int raise(int sig);
```

Description

`raise` sends the signal *sig* to the running program. You can then use signal to handle *sig*.

Signals and signal handling are described in signal, and in the *VisualAge C++ Programming Guide* under "Signal and Exception Handling".

Returns

raise returns `0` if successful, nonzero if unsuccessful.

Example Code

This example establishes a signal handler called `sig_hand` for the signal SIGUSR1. The signal handler is called whenever the SIGUSR1 signal is raised and will ignore the first nine occurrences of the signal. On the tenth raised signal, it exits the program with an error code of 10. Note that the signal handler must be reestablished each time it is called.

```
#include <signal.h>
#include <stdio.h>

void sig_hand(int);        /* declaration of sig_hand() as a function         */
int main(void)
{
   signal(SIGUSR1, sig_hand);              /* set up handler for SIGUSR1      */

   raise(SIGUSR1);                      /* signal SIGUSR1 is raised           */
                       /* sig_hand() is called                                */
   return 0;
}

void sig_hand(int sig)
{
   static int count = 0;                 /* initialized only once             */
   count++;

   if (10 == count)     /* ignore the first 9 occurrences of this signal      */
      exit(10);
   else
      signal(SIGUSR1, sig_hand);               /* set up the handler again    */
   raise(SIGUSR1);
}
```

Related Information

- signal
- "Signal and Exception Handling" in the *VisualAge C++ Programming Guide*

----

rand - Generate Random Number

rand - Generate Random Number

Syntax

```
#include <stdlib.h>
int rand(void);
```

Description

`rand` generates a pseudo-random integer in the range `0` to `RAND_MAX` (macro defined in `<stdlib.h>`). Use `srand` before calling `rand` to set a starting point for the random number generator. If you do not call `srand` first, the default seed is 1.

Returns

`rand` returns a pseudo-random number.

Example Code

This example prints the first 10 random numbers generated.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("iteration %d, rand=%d\n", x, rand());
   return 0;

   /*****************************************************************************
      The output should be:

      iteration 1, rand=16838
      iteration 2, rand=5758
      iteration 3, rand=10113
      iteration 4, rand=17515
      iteration 5, rand=31051
      iteration 6, rand=5627
      iteration 7, rand=23010
      iteration 8, rand=7419
      iteration 9, rand=16212
      iteration 10, rand=4086
   *****************************************************************************/
}
```

Related Information

- srand

----

read - Read Into Buffer

read - Read Into Buffer

Syntax

```
#include <io.h>
int read(int handle, char *buffer, unsigned int count);
```

Description

read reads *count* bytes from the file associated with *handle* into *buffer* . The read operation begins at the current position of the file pointer associated with the given file. After the read operation, the file pointer points to the next unread character.

Note: In earlier releases of C Set ++, read began with an underscore (`_read`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_read` to read for you.

Returns

read returns the number of bytes placed in *buffer* . This number can be less than *count* if there are fewer than *count* bytes left in the file or if the file was opened in text mode. (See the note below.) The return value `0` indicates an attempt to read at end-of-file. A return value `-1` indicates an error. If `-1` is returned, the current file position is undefined, and `errno` is set to one of the following values: compact break=fit.

Value	Meaning
EBADF	The given handle is incorrect, or the file is not open for reading, or the file is locked.
EOS2ERR	The call to the operating system was not successful.

Note: If the file was opened in text mode, the return value might not correspond to the number of bytes actually read. When text mode is in effect, carriage return characters are deleted from the input stream without affecting the file pointer.

Example Code

This example opens the file `sample.dat` and attempts to read from it.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
   int fh;
   char buffer[20];

   memset(buffer, '\0', 20);
   printf("\nCreating sample.dat.\n");
   system("echo Sample Program > sample.dat");
   if (-1 == (fh = open("sample.dat", O_RDWR|O_APPEND))) {
      perror("Unable to open sample.dat.");
      return EXIT_FAILURE;
   }
   if (7 != read(fh, buffer, 7)) {
      perror("Unable to read from sample.dat.");
      close(fh);
      return EXIT_FAILURE;
   }
   printf("Successfully read in the following:\n%s\n ", buffer);
   close(fh);
   return 0;

   /****************************************************************************
      The output should be:

      Creating sample.dat.
      Successfully read in the following:
      Sample
   ****************************************************************************/
}
```

Related Information

- creat
- fread
- open
- _sopen
- write

----

realloc - Change Reserved Storage Block Size

realloc - Change Reserved Storage Block Size

Syntax

```
#include <stdlib.h>  /* also in <malloc.h> */
void *realloc(void *ptr, size_t size);
```

Description

`realloc` changes the size of a previously reserved storage block. The *ptr* argument points to the beginning of the block. The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes. realloc allocates the new block from the same heap the original block was in.

If *ptr* is `NULL`, realloc reserves a block of storage of *size* bytes from the current thread's default heap (equivalent to calling `malloc(`*size*`)`).

If *size* is `0` and the *ptr* is not `NULL`, realloc frees the storage allocated to *ptr* and returns `NULL`

Returns

`realloc` returns a pointer to the reallocated storage block. The storage location of the block may be moved by the `realloc` function. Thus, the *ptr* argument to `realloc` is not necessarily the same as the return value.

If *size* is `0`, realloc returns `NULL`. If there is not enough storage to expand the block to the given size, the original block is unchanged and realloc returns `NULL`.

The storage to which the return value points is aligned for storage of any type of object.

Example Code

This example allocates storage for the prompted size of `array` and then uses `realloc` to reallocate the block to hold the new size of the array. The contents of the array are printed after each allocation.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long *array;                         /* start of the array           */
   long *ptr;                           /* pointer to array */
   int i;                               /* index variable   */
   int num1,num2;                       /* number of entries of the array     */
   void print_array(long *ptr_array, int size);

   printf("Enter the size of the array\n");
   scanf("%i", &num1);
   /* allocate num1 entries using malloc()                                    */

   if ((array = malloc(num1 *sizeof(long))) != NULL) {
      for (ptr = array, i = 0; i < num1; ++i)             /* assign values    */
         *ptr++ = i;
      print_array(array, num1);
      printf("\n");
   }
   else {                                                 /* malloc error     */
      perror("Out of storage");
      abort();
   }

   /* Change the size of the array ...                                        */

   printf("Enter the size of the new array\n");
   scanf("%i", &num2);
   if ((array = realloc(array, num2 *sizeof(long))) != NULL) {
      for (ptr = array+num1, i = num1; i <= num2; ++i)
         *ptr++ = i+2000;                /* assign values to new elements     */
      print_array(array, num2);
   }
   else {                                                 /* realloc error    */
      perror("Out of storage");
      abort();
   }
   return 0;

   /*****************************************************************************
      The output should be similar to:

      Enter the size of the array
      2
      The array of size 2 is:
        array[ 0 ] = 0
        array[ 1 ] = 1
      Enter the size of the new array
      4
      The array of size 4 is:
        array[ 0 ] = 0
        array[ 1 ] = 1
        array[ 2 ] = 2002
        array[ 3 ] = 2003
   *****************************************************************************/
}


void print_array(long *ptr_array,int size)
{
   int i;
   long *index = ptr_array;

   printf("The array of size %d is:\n", size);
   for (i = 0; i < size; ++i)                             /* print the array
                                                             out              */
      printf("  array[ %i ] = %li\n", i, ptr_array[i]);
}
```

Related Information

- calloc
- free
- malloc
- _msize

----

regcomp - Compile Regular Expression

regcomp - Compile Regular Expression

Syntax

```
#include <sys/types.h>
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

Description

regcomp compiles the source regular expression pointed to by *pattern* into an executable version and stores it in the location pointed to by *preg* . You can then use regexec to compare the regular expression to other strings.

The *cflags* flag defines the attributes of the compilation process:

REG_EXTENDED
Support extended regular expressions.

REG_ICASE
Ignore case in match.

REG_NEWLINE
Treat new-line character as a special end-of-line character; it then establishes the line boundaries matched by the ^ and $ patterns, and can only be matched within a string explicitly using \n. (If you omit this flag, the new-line character is treated like any other character.)

REG_NOSUB
Ignore the number of subexpressions specified in *pattern* . When you compare a string to the compiled pattern (using regexec), the string must match the entire pattern. regexec then returns a value that indicates only if a match was found; it does not indicate at what point in the string the match begins, or what the matching string is.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, which can be interpreted differently depending on the current locale. The functions regcomp, regerror, regexec, and regfree use regular expressions in a similar way to the UNIX **awk**, **ed**, **grep**, and **egrep** commands. Regular expressions are described in more detail under "Regular Expressions" in the *VisualAge C++ Programming* Guide.

Returns

If regcomp is successful, it returns 0. Otherwise, it returns an error code that you can use in a call to regerror, and the content of *preg* is undefined.

Example Code

This example compiles an extended regular expression.

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t preg;
   char    *pattern = ".*(simple).*";
   int     rc;

   if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
      printf("regcomp() failed, returning nonzero (%d)\n", rc);
      exit(EXIT_FAILURE);
   }
   printf("regcomp() is sucessful.\n");
   return 0;

   /****************************************************************************
      The output should be similar to :

      regcomp() is sucessful.
   ****************************************************************************/
}
```

Related Information

- regerror
- regexec
- regfree

----

regerror - Return Error Message for Regular Expression

regerror - Return Error Message for Regular Expression

Syntax

```
#include <sys/types.h>
#include <regex.h>
size_t regerror(int errcode,  const regex_t *preg,
                char *errbuf, size_t errbuf_size);
```

Description

regerror finds the description for the error code *errcode* for the regular expression *preg*. The description for *errcode* is assigned to *errbuf*. *errbuf_size* is a value that you provide that specifies the maximum message size that can be stored (the size of *errbuf*).

Regular expressions are described in detail under "Regular Expressions" in the *VisualAge C++ Programming Guide*.

The description strings for *errcode* are:

errcode	**Description String**
REG_NOMATCH	RE pattern not found
REG_BADPAT	Invalid regular expression
REG_ECOLLATE	Invalid collating element
REG_ECTYPE	Invalid character class
REG_EESCAPE	Last character is \
REG_ESUBREG	Invalid number in \digit
REG_EBRACK	[] imbalance
REG_EPAREN	\(\) or () imbalance
REG_EBRACE	\{ \} or { } imbalance
REG_BADBR	Invalid \{ \} range exp
REG_ERANGE	Invalid range exp endpoint
REG_ESPACE	Out of memory
REG_BADRPT	?*+ not preceded by valid RE
REG_ECHAR	Invalid multibyte character
REG_EBOL	^ anchor and not BOL
REG_EEOL	$ anchor and not EOL

Returns

regerror returns the size of the buffer needed to hold the string that describes the error condition.

Example Code

This example compiles an invalid regular expression, and prints an error message using regerror.

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t preg;
   char    *pattern = "a[missing.bracket";
   int     rc;
   char    buffer[100];

   if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
      regerror(rc, &preg, buffer, 100);
      printf("regcomp() failed with '%s'\n", buffer);
      exit(EXIT_FAILURE);
   }
   return 0;

   /****************************************************************************
      The output should be similar to :

      regcomp() failed with '[] imbalance'
   ****************************************************************************/
}
```

Related Information

- regcomp
- regexec
- regfree

----

regexec - Execute Compiled Regular Expression

regexec - Execute Compiled Regular Expression

Syntax

```
#include <sys/types.h>
#include <regex.h>
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```

Description

regexec compares the null-terminated *string* against the compiled regular expression *preg* to find a match between the two. (Regular expressions are described in "Regular Expressions" in the *VisualAge C++ Programming Guide* .)

nmatch is the number of substrings in *string* that regexec should try to match with subexpressions in *preg* . The array you supply for *pmatch* must have at least *nmatch* elements.

regexec fills in the elements of the array *pmatch* with offsets of the substrings in *string* that correspond to the parenthesized subexpressions of the original pattern given to regcomp to create *preg* . The zeroth element of the array corresponds to the entire pattern. If there are more than *nmatch* subexpressions, only the first *nmatch* - 1 are stored. If *nmatch* is 0, or if the REG_NOSUB flag was set when *preg* was created with regcomp, regexec ignores the *pmatch* argument.

The *eflags* flag defines customizable behavior of regexec:

REG_NOTBOL
: Indicates that the first character of *string* is not the beginning of line.

REG_NOTEOL
: Indicates that the first character of *string* is not the end of line.

When a basic or extended regular expression is matched, any given parenthesized subexpression of the original pattern could participate in the match of several different substrings of *string* . The following rules determine which substrings are reported in *pmatch.* :

1. If a subexpression participated in a match several times, regexec stores the offset of the last matching substring in *pmatch* .

2. If a subexpression did not match in the source *string* , regexec sets the offset shown in *pmatch* to -1.

3. If a subexpression contains subexpressions, the data in *pmatch* refers to the last such subexpression.

4. If a subexpression matches a zero-length string, the offsets in *pmatch* refer to the byte immediately following the matching string.

If the REG_NOSUB flag was set when *preg* was created by regcomp, the contents of *pmatch* are unspecified. If the REG_NEWLINE flag was not set when *preg* was created, new-line characters are allowed in *string* .

Returns

If a match is found, regexec returns 0. Otherwise, it returns a nonzero value indicating either no match or an error.

Example Code

This example compiles an expression and matches a string against it. The first substring uses the full pattern. The second substring uses the sub-expression inside the full pattern.

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t    preg;
   char       *string = "a very simple simple simple string";
   char       *pattern = "\\(sim[a-z]le\\) \\1";
   int        rc;
   size_t     nmatch = 2;
   regmatch_t pmatch[2];

   if (0 != (rc = regcomp(&preg, pattern, 0))) {
      printf("regcomp() failed, returning nonzero (%d)\n", rc);
      exit(EXIT_FAILURE);
   }

   if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
      printf("Failed to match '%s' with '%s',returning %d.\n",
      string, pattern, rc);
   }
   else {
      printf("With the whole expression, "
             "a matched substring \"%.*s\" is found at position %d to %d.\n",
             pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
             pmatch[0].rm_so, pmatch[0].rm_eo - 1);
      printf("With the sub-expression, "
             "a matched substring \"%.*s\" is found at position %d to %d.\n",
             pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
             pmatch[1].rm_so, pmatch[1].rm_eo - 1);
   }
   regfree(&preg);
   return 0;

   /****************************************************************************
      The output should be similar to :

      With the whole expression, a matched substring "simple simple" is found
      at position 7 to 19.
      With the sub-expression, a matched substring "simple" is found
      at position 7 to 12.
   ****************************************************************************/
}
```

Related Information

- regcomp
- regerror
- regfree

----

regfree - Free Memory for Regular Expression

regfree - Free Memory for Regular Expression

Syntax

```
#include <sys/types.h>
#include <regex.h>
void regfree(regex_t *preg);
```

Description

regfree frees any memory that was allocated by regcomp to implement the regular expression *preg* . After the call to regfree, the expression defined by *preg* is no longer a compiled regular or extended expression.

Regular expressions are described in "Regular Expressions" in the *VisualAge C++ Programming Guide* .

Returns

There is no return value.

Example Code

This example compiles an extended regular expression and frees it.

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   regex_t    preg;
   char       *pattern = ".*(simple).*";
   int        rc;

   if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
      printf("regcomp() failed, returning nonzero (%d)\n", rc);
      exit(EXIT_FAILURE);
   }
   regfree(&preg);
   printf("Memory allocated for reg is freed.\n");
   return 0;

   /****************************************************************************
      The output should be similar to :

      Memory allocated for reg is freed.
   ****************************************************************************/
}
```

Related Information

- regcomp
- regerror
- regexec

----

remove - Delete File

remove - Delete File

Syntax

```
#include <stdio.h>
int remove(const char *filename);
```

Description

`remove` deletes the file specified by *filename* .

Note: You cannot remove a nonexistent file or a file that is open.

Returns

`remove` returns `0` if it successfully deletes the file. A nonzero return value idicates an error.

Example Code

This example uses `remove` to remove a file. It issues a message if an error occurs.

```
#include <stdio.h>

int main(void)
{
   char *FileName = "file2rm.mjq";
   FILE *fp;

   fp = fopen(FileName, "w");
   fprintf(fp, "Hello world\n");
   fclose(fp);
   if (remove(FileName) != 0)
      perror("Could not remove file");
   else
      printf("File \"%s\" removed successfully.\n", FileName);
   return 0;

   /****************************************************************************
      The output should be:

      File "file2rm.mjq" removed successfully.
   ****************************************************************************/
}
```

Related Information

- fopen
- rename
- _rmtmp
- unlink

----

rename - Rename File

rename - Rename File

Syntax

```
#include <stdio.h>  /* also in <io.h> */
int rename(const char *oldname, const char *newname);
```

Description

`rename` renames the file specified by *oldname* to the name given by *newname* . The *oldname* pointer must specify the name of an existing file. The *newname* pointer must not specify the name of an existing file. Both *oldname* and *newname* must be on the same drive; you cannot rename files across drives. You cannot rename a file with the name of an existing file. You also cannot rename an open file.

Returns

`rename` returns `0` if successful. On an error, it returns a nonzero value.

Example Code

This example uses `rename` to rename a file. It issues a message if errors occur.

```
#include <stdio.h>

int main(void)
{
   char *OldName = "oldfile.mjq";
   char *NewName = "newfile.mjq";
   FILE *fp;

   fp = fopen(OldName, "w");
   fprintf(fp, "Hello world\n");
   fclose(fp);
   if (rename(OldName, NewName) != 0)
      perror("Could not rename file");
   else
      printf("File \"%s\" is renamed to \"%s\".\n", OldName, NewName);
   return 0;

   /*****************************************************************************
      The output should be:

      File "oldfile.mjq" is renamed to "newfile.mjq".
   *****************************************************************************/
}
```

Related Information

- fopen
- remove

----

rewind - Adjust Current File Position

rewind - Adjust Current File Position

Syntax

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

`rewind` repositions the file pointer associated with *stream* to the beginning of the file. A call to `rewind` is the same as:

```
   (void)fseek(stream, 0L, SEEK_SET);
```

except that `rewind` also clears the error indicator for the *stream* .

Returns

There is no return value.

Example Code

This example first opens a file myfile.dat for input and output. It writes integers to the file, uses `rewind` to reposition the file pointer to the beginning of the file, and then reads the data back in.

```
#include <stdio.h>

FILE *stream;
int data1,data2,data3,data4;

int main(void)
{
   data1 = 1;
   data2 = -37;

       /* Place data in the file                                              */

   stream = fopen("myfile.dat", "w+");
   fprintf(stream, "%d %d\n", data1, data2);

       /* Now read the data file                                              */

   rewind(stream);
   fscanf(stream, "%d", &data3);
   fscanf(stream, "%d", &data4);
   printf("The values read back in are: %d and %d\n", data3, data4);
   return 0;

   /****************************************************************************
      The output should be:

      The values read back in are: 1 and -37
   ****************************************************************************/
}
```

Related Information

- fgetpos
- fseek
- fsetpos
- ftell
- lseek
- _tell

----

rmdir - Remove Directory

rmdir - Remove Directory

Syntax

```
#include <direct.h>
int rmdir(char *pathname);
```

Description

rmdir deletes the directory specified by *pathname* . The directory must be empty, and it must not be the current working directory or the root directory.

Note: In earlier releases of C Set ++, rmdir began with an underscore (_rmdir). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map _rmdir to rmdir for you.

Returns

rmdir returns the value 0 if the directory is successfully deleted. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	**Meaning**
EACCESS	One of the following has occurred: • The given path name is not a directory. • The directory is not empty. • The directory is read only. • The directory is the current working directory or root directory being used by a process.
ENOENT	The path name was not found.

Example Code

This example deletes two directories: one in the root directory, and the other in the current working directory.

```
#include <stdio.h>
#include <direct.h>
#include <string.h>

int main(void)
{
   char *dir1,*dir2;

 /*  Create the directory "aleng" in the root directory of the C: drive.      */

   dir1 = "c:\\aleng";
   if (0 == (mkdir(dir1)))
      printf("%s directory was created.\n", dir1);
   else
      printf("%s directory was not created.\n", dir1);

 /*  Create the subdirectory "simon" in the current directory.                */

   dir2 = "simon";
   if (0 == (mkdir(dir2)))
      printf("%s directory was created.\n", dir2);
   else
      printf("%s directory was not created.\n", dir2);

 /*  Remove the directory "aleng" from the root directory of the C: drive.    */

   printf("Removing directory 'aleng' from the root directory.\n");
   if (rmdir(dir1))
      perror(NULL);
   else
      printf("%s directory was removed.\n", dir1);

 /*  Remove the subdirectory "simon" from the current directory.              */

   printf("Removing subdirectory 'simon' from the current directory.\n");
   if (rmdir(dir2))
      perror(NULL);
   else
      printf("%s directory was removed.\n", dir2);
   return 0;

   /***************************************************************************
      The output should be:

      c:\aleng directory was created.
      simon directory was created.
      Removing directory 'aleng' from the root directory.
      c:\aleng directory was removed.
      Removing subdirectory 'simon' from the current directory.
      simon directory was removed.
   ****************************************************************************/
}
```

Related Information

- chdir
- _getdcwd
- _getcwd
- mkdir

_rmem_init - Initialize Memory Functions for Subsystem DLL

_rmem_init - Initialize Memory Functions for Subsystem DLL

Syntax

```
int _rmem_init(void);
/* no header file - defined in run-time startup code */
```

Description

_rmem_init initializes the memory functions for subsystem DLLs. Although subsystems do not require a run-time environment (and therefore do not call _CRT_init), they do require the library memory functions. For DLLs that do use a run-time environment, the memory functions are initialized with the environment by _CRT_init.

By default, all DLLs call *The Developer's Toolkit* _DLL_InitTerm function, which in turn calls _rmem_init for you. However, if you are writing your own subsystem _DLL_InitTerm function (for example, to perform actions other than memory initialization and termination), you must call _rmem_init from your version of _DLL_InitTerm before you can call any other library functions.

If your DLL contains C++ code, you must also call `__ctordtorInit` after _rmem_init to ensure that static constructors and destructors are initialized properly. __ctordtorInit is defined in the run-time startup code as:

```
   void __ctordtorInit(void);
```

Returns

If the memory functions were successfully initialized, _rmem_init returns `0`. A return code of `-1` indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of stderr.

Example Code

The following example shows the _DLL_InitTerm function from *The Developer's Toolkit* sample program for building subsystem DLLs, which calls _rmem_init to initialize the memory functions.

```
#pragma strings( readonly )
/******************************************************************************/
/*                                                                            */
/* _DLL_InitTerm - Initialization/Termination function for the DLL that is    */
/*                 invoked by the loader.                                     */
/*                                                                            */
/* DLLREGISTER  - Called by _DLL_InitTerm for each process that loads the     */
/*                DLL.                                                        */
/*                                                                            */
/* DLLDEREGISTER- Called by _DLL_InitTerm for each process that frees the     */
/*                DLL.                                                        */
/*                                                                            */
/******************************************************************************/

#define  INCL_DOS
#define  INCL_DOSERRORS
#define  INCL_NOPMAPI
#include <os2.h>
#include <stdio.h>

unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag );

static unsigned long DLLREGISTER( void );
static unsigned long DLLDEREGISTER( void );

#define SHARED_SEMAPHORE_NAME "\\SEM32\\SAMPLE05\\DLL.LCK"

/* The following data will be per-process data.  It will not be shared among  */
/* different processes.                                                       */

static HMTX  hmtxSharedSem;    /* Shared semaphore                            */
static ULONG ulProcessTotal;   /* Total of increments for a process           */
static PID   pidProcess;       /* Process identifier                          */

/* This is the global data segment that is shared by every process.           */

#pragma data_seg( GLOBAL_SEG )

static ULONG ulProcessCount;                /* total number of processes      */

/* _DLL_InitTerm() - called by the loader for DLL initialization/termination  */
/* This function must return a non-zero value if successful and a zero value  */
/* if unsuccessful.                                                           */
unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag )
   {
   APIRET rc;




   /* If ulFlag is zero then initialization is required:                      */
   /*    If the shared memory pointer is NULL then the DLL is being loaded    */
   /*    for the first time so acquire the named shared storage for the       */
   /*    process control structures.  A linked list of process control        */
   /*    structures will be maintained.  Each time a new process loads this   */
   /*    DLL, a new process control structure is created and it is inserted   */
   /*    at the end of the list by calling DLLREGISTER.                       */
   /*                                                                         */
   /* If ulFlag is 1 then termination is required:                            */
   /*    Call DLLDEREGISTER which will remove the process control structure   */
   /*    and free the shared memory block from its virtual address space.     */

   switch( ulFlag )
      {
   case 0:
         if ( !ulProcessCount )
            {
            _rmem_init();
            /* Create the shared mutex semaphore.                             */
            if ( ( rc = DosCreateMutexSem( SHARED_SEMAPHORE_NAME,
                                           &hmtxSharedSem,
                                           0,
                                           FALSE ) ) != NO_ERROR )
               {
               printf( "DosCreateMutexSem rc = %lu\n", rc );
               return 0;
               }
            }

         /* Register the current process.                                     */
         if ( DLLREGISTER( ) )
            return 0;
         break;

      case 1:
         /* De-register the current process.                                  */
         if ( DLLDEREGISTER( ) )
            return 0;

         _rmem_term();
         break;
      default:
         return 0;
      }

   /* Indicate success.  Non-zero means success!!!                            */
   return 1;
   }




/* DLLREGISTER - Registers the current process so that it can use this        */
/*               subsystem.  Called by _DLL_InitTerm when the DLL is first    */
/*               loaded for the current process.                              */

static unsigned long DLLREGISTER( void )
   {
   APIRET rc;
   PTIB ptib;
   PPIB ppib;

   /* Get the address of the process and thread information blocks.           */
   if ( ( rc = DosGetInfoBlocks( &ptib, &ppib ) ) != NO_ERROR )
      {
      printf( "DosGetInfoBlocks rc = %lu\n", rc );
      return rc;
      }

   /* Open the shared mutex semaphore for this process.                       */
   if ( ( rc = DosOpenMutexSem( SHARED_SEMAPHORE_NAME,
                                &hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosOpenMutexSem rc = %lu\n", rc );
      return rc;
      }

   /* Acquire the shared mutex semaphore.                                     */
   if ( ( rc = DosRequestMutexSem( hmtxSharedSem,
                                   SEM_INDEFINITE_WAIT ) ) != NO_ERROR )
      {
      printf( "DosRequestMutexSem rc = %lu\n", rc );
      DosCloseMutexSem( hmtxSharedSem );
      return rc;
      }

   /* Increment the count of processes registered.                            */
   ++ulProcessCount;

   /* Initialize the per-process data.                                        */
   ulProcessTotal = 0;
   pidProcess = ppib->pib_ulpid;

   /* Tell the user that the current process has been registered.             */
   printf( "\nProcess %lu has been registered.\n\n", pidProcess );

   /* Release the shared mutex semaphore.                                     */
   if ( ( rc = DosReleaseMutexSem( hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosReleaseMutexSem rc = %lu\n", rc );
      return rc;
      }

   return 0;
   }




/* DLLDEREGISTER - Deregisters the current process from this subsystem.       */
/*                 Called by _DLL_InitTerm when the DLL is freed for the      */
/*                 last time by the current process.                          */

static unsigned long DLLDEREGISTER( void )
   {
   APIRET rc;

   /* Acquire the shared mutex semaphore.                                     */
   if ( ( rc = DosRequestMutexSem( hmtxSharedSem,
                                   SEM_INDEFINITE_WAIT ) ) != NO_ERROR )
      {
      printf( "DosRequestMutexSem rc = %lu\n", rc );
      return rc;
      }
   /* Decrement the count of processes registered.                            */
   --ulProcessCount;

   /* Tell the user that the current process has been deregistered.           */
   printf( "\nProcess %lu has been deregistered.\n\n", pidProcess );

   /* Release the shared mutex semaphore.                                     */
   if ( ( rc = DosReleaseMutexSem( hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosReleaseMutexSem rc = %lu\n", rc );
      return rc;
      }

   /* Close the shared mutex semaphore for this process.                      */
   if ( ( rc = DosCloseMutexSem( hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosCloseMutexSem rc = %lu\n", rc );
      return rc;
      }

   return 0;
   }
```

Related Information

- "Building Subsystem DLLs" in the *VisualAge C++ Programming Guide*
- _CRT_init
- _CRT_term
- _DLL_InitTerm
- _rmem_term

_rmem_term - Terminate Memory Functions for Subsystem DLL

_rmem_term - Terminate Memory Functions for Subsystem DLL

Syntax

```
int _rmem_term(void);
/* no header file - defined in run-time startup code */
```

Description

_rmem_term terminates the memory functions for subsystem DLLs. It is only needed for DLLs where the run-time library is statically linked.

By default, all DLLs call *The Developer's Toolkit* _DLL_InitTerm function, which in turn calls _rmem_term for you. However, if you are writing your own _DLL_InitTerm function (for example, to perform actions other than memory initialization and termination), and your DLL statically links to the C run-time libraries, you need to call _rmem_term from your subsystem _DLL_InitTerm function. (For DLLs with a run-time environment, this termination is done by _CRT_term.)

If your DLL contains C++ code, you must also call `__ctordtorTerm` **before** you call _rmem_term to ensure that static constructors and destructors are terminated correctly. __ctordtorTerm is defined in the run-time startup code as:

```
   void __ctordtorTerm(void);
```

Once you have called _rmem_term, you cannot call any other library functions.

If your DLL is dynamically linked, you cannot call library functions in the termination section of your _DLL_InitTerm function. If your termination routine requires calling library functions, you must register the termination routine with `DosExitList`. Note that all `DosExitList` routines are called before DLL termination routines.

Returns

_rmem_term returns `0` if the memory functions were successfully terminated. A return value of `-1` indicates an error.

Example Code

The following example shows the _DLL_InitTerm function from *The Developer's Toolkit* sample program for building subsystem DLLs, which calls _rmem_term to terminate the library memory functions.

```
#pragma strings( readonly )
/******************************************************************************/
/*                                                                            */
/* _DLL_InitTerm - Initialization/Termination function for the DLL that is    */
/*                 invoked by the loader.                                     */
/*                                                                            */
/* DLLREGISTER  - Called by _DLL_InitTerm for each process that loads the     */
/*                DLL.                                                        */
/*                                                                            */
/* DLLDEREGISTER- Called by _DLL_InitTerm for each process that frees the     */
/*                DLL.                                                        */
/*                                                                            */
/******************************************************************************/

#define  INCL_DOS
#define  INCL_DOSERRORS
#define  INCL_NOPMAPI
#include <os2.h>
#include <stdio.h>

unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag );

static unsigned long DLLREGISTER( void );
static unsigned long DLLDEREGISTER( void );

#define SHARED_SEMAPHORE_NAME "\\SEM32\\SAMPLE05\\DLL.LCK"

/* The following data will be per-process data.  It will not be shared among  */
/* different processes.                                                       */

static HMTX  hmtxSharedSem;    /* Shared semaphore                            */
static ULONG ulProcessTotal;   /* Total of increments for a process           */
static PID   pidProcess;       /* Process identifier                          */

/* This is the global data segment that is shared by every process.           */

#pragma data_seg( GLOBAL_SEG )

static ULONG ulProcessCount;                /* total number of processes      */

/* _DLL_InitTerm() - called by the loader for DLL initialization/termination  */
/* This function must return a non-zero value if successful and a zero value  */
/* if unsuccessful.                                                           */
unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag )
   {
   APIRET rc;




   /* If ulFlag is zero then initialization is required:                      */
   /*    If the shared memory pointer is NULL then the DLL is being loaded    */
   /*    for the first time so acquire the named shared storage for the       */
   /*    process control structures.  A linked list of process control        */
   /*    structures will be maintained.  Each time a new process loads this   */
   /*    DLL, a new process control structure is created and it is inserted   */
   /*    at the end of the list by calling DLLREGISTER.                       */
   /*                                                                         */
   /* If ulFlag is 1 then termination is required:                            */
   /*    Call DLLDEREGISTER which will remove the process control structure   */
   /*    and free the shared memory block from its virtual address space.     */

   switch( ulFlag )
      {
   case 0:
         if ( !ulProcessCount )
            {
            _rmem_init();
            /* Create the shared mutex semaphore.                             */
            if ( ( rc = DosCreateMutexSem( SHARED_SEMAPHORE_NAME,
                                           &hmtxSharedSem,
                                           0,
                                           FALSE ) ) != NO_ERROR )
               {
               printf( "DosCreateMutexSem rc = %lu\n", rc );
               return 0;
               }
            }

         /* Register the current process.                                     */
         if ( DLLREGISTER( ) )
            return 0;
         break;

      case 1:
         /* De-register the current process.                                  */
         if ( DLLDEREGISTER( ) )
            return 0;

         _rmem_term();
         break;

      default:
         return 0;
      }

   /* Indicate success.  Non-zero means success!!!                            */

   return 1;
   }

/* DLLREGISTER - Registers the current process so that it can use this        */
/*               subsystem.  Called by _DLL_InitTerm when the DLL is first    */
/*               loaded for the current process.                              */

static unsigned long DLLREGISTER( void )
   {
   APIRET rc;
   PTIB ptib;
   PPIB ppib;




   /* Get the address of the process and thread information blocks.           */
   if ( ( rc = DosGetInfoBlocks( &ptib, &ppib ) ) != NO_ERROR )
      {
      printf( "DosGetInfoBlocks rc = %lu\n", rc );
      return rc;
      }

   /* Open the shared mutex semaphore for this process.                       */
   if ( ( rc = DosOpenMutexSem( SHARED_SEMAPHORE_NAME,
                                &hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosOpenMutexSem rc = %lu\n", rc );
      return rc;
      }

   /* Acquire the shared mutex semaphore.                                     */
   if ( ( rc = DosRequestMutexSem( hmtxSharedSem,
                                   SEM_INDEFINITE_WAIT ) ) != NO_ERROR )
      {
      printf( "DosRequestMutexSem rc = %lu\n", rc );
      DosCloseMutexSem( hmtxSharedSem );
      return rc;
      }

   /* Increment the count of processes registered.                            */
   ++ulProcessCount;

   /* Initialize the per-process data.                                        */
   ulProcessTotal = 0;
   pidProcess = ppib->pib_ulpid;

   /* Tell the user that the current process has been registered.             */
   printf( "\nProcess %lu has been registered.\n\n", pidProcess );

   /* Release the shared mutex semaphore.                                     */
   if ( ( rc = DosReleaseMutexSem( hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosReleaseMutexSem rc = %lu\n", rc );
      return rc;
      }

   return 0;
   }




/* DLLDEREGISTER - Deregisters the current process from this subsystem.       */
/*                 Called by _DLL_InitTerm when the DLL is freed for the      */
/*                 last time by the current process.                          */

static unsigned long DLLDEREGISTER( void )
   {
   APIRET rc;

   /* Acquire the shared mutex semaphore.                                     */
   if ( ( rc = DosRequestMutexSem( hmtxSharedSem,
                                   SEM_INDEFINITE_WAIT ) ) != NO_ERROR )
      {
      printf( "DosRequestMutexSem rc = %lu\n", rc );
      return rc;
      }

   /* Decrement the count of processes registered.                            */
   --ulProcessCount;

   /* Tell the user that the current process has been deregistered.           */
   printf( "\nProcess %lu has been deregistered.\n\n", pidProcess );

   /* Release the shared mutex semaphore.                                     */
   if ( ( rc = DosReleaseMutexSem( hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosReleaseMutexSem rc = %lu\n", rc );
      return rc;
      }

   /* Close the shared mutex semaphore for this process.                      */
   if ( ( rc = DosCloseMutexSem( hmtxSharedSem ) ) != NO_ERROR )
      {
      printf( "DosCloseMutexSem rc = %lu\n", rc );
      return rc;
      }

   return 0;
   }
```

Related Information

- "Building Subsystem DLLs" in the *VisualAge C++ Programming Guide*
- _CRT_init
- _CRT_term
- _DLL_InitTerm
- _rmem_init

_rmtmp - Remove Temporary Files

_rmtmp - Remove Temporary Files

Syntax

```
#include <stdio.h>
int _rmtmp(void);
```

Description

_rmtmp closes and deletes all temporary files in all directories that are held open by the calling process.

Returns

_rmtmp returns the number of temporary files deleted.

Example Code

This example uses _rmtmp to remove a number of temporary files.

```
#include <stdio.h>

int main(void)
{
   int num;
   FILE *stream;

   if (NULL == (stream = tmpfile()))
      printf("Could not open new temporary file\n");
   else {
      num = _rmtmp();
      printf("Number of temporary files removed = %d\n", num);
   }
   return 0;

   /*****************************************************************************
      The output should be:

      Number of temporary files removed = 1
   *****************************************************************************/
}
```

Related Information

- _flushall
- remove
- tmpfile
- tmpnam
- unlink

----

scanf - Read Data

scanf - Read Data

Syntax

```
#include <stdio.h>
int scanf(const char *format-string, argument-list);
```

Description

`scanf` reads data from the standard input stream stdin into the locations given by each entry in *argument-list*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format-string*. The *format-string* controls the interpretation of the input fields.

The *format-string* can contain one or more of the following:

- White-space characters, as specified by `isspace` (such as blanks and new-line characters). A white-space character causes `scanf` to read, but not to store, all consecutive white-space characters in the input up to the next character that is not white space. One white-space character in *format-string* matches any combination of white-space characters in the input.

- Characters that are not white space, except for the percent sign character (%). A non-white-space character causes `scanf` to read, but not to store, a matching non-white-space character. If the next character in stdin does not match, `scanf` ends.

- Format specifications, introduced by the percent sign (%). A format specification causes `scanf` to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

`scanf` reads *format-string* from left to right. Characters outside of format specifications are expected to match the sequence of characters in stdin; the matched characters in stdin are scanned but not stored. If a character in stdin conflicts with *format-string* , `scanf` ends. The conflicting character is left in stdin as if it had not been read.

When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the first entry in *argument-list* . The second format specification converts the second input field and stores it in the second entry in *argument-list* , and so on through the end of *format-string* .

An input field is defined as all characters up to the first white-space character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until the field *width* is reached, whichever comes first. If there are too many arguments for the format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

A format specification has the following form:

```
>>  %                                  type  ><
        n$     *     width     h
                                  l
                                  L
```

In the format specification above, `%n$` specifies the nth argument.

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, `%s`). Each field of the format specification is discussed in detail below.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *type* . The field is scanned but not stored.

The *width* is a positive decimal integer controlling the maximum number of characters to be read from stdin. No more than *width* characters are converted and stored at the corresponding *argument* . Fewer than *width* characters are read if a white-space character (space, tab, or new line), or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional prefix `l` shows that you use the **long** version of the following *type* , while the prefix `h` indicates that the **short** version is to be used. The corresponding *argument* should point to a **long** or **double** object (for the `l` character), a **long double** object (for the `L` character), or a **short** object (with the `h` character). The `l` and `h` modifiers can be used with the `d`, `i`, `o`, `x`, and `u` *type* characters. The `l` modifier can also be used with the `e`, `f`, and `g` *type* characters. The `L` modifier can be used with the `e`, `f` and `g` *type* characters. The `l` and `h` modifiers are ignored if specified for any other *type* . Note that the `l` modifier is also used with the `c` and `s` characters to indicate a multibyte character or string.

The *type* characters and their meanings are in the following table:

Character	Type of Input Expected.	Type of Argument.
"d"	Decimal integer.	Pointer to INT.
"o"	Octal integer.	Pointer to UNSIGNED INT.
"x", "X"	Hexadecimal integer.	Pointer to UNSIGNED INT.
"i"	Decimal, hexadecimal, or octal integer.	Pointer to INT.

"u"	Unsigned decimal integer.	Pointer to UNSIGNED INT.
"e, f, g" "E, G"	Floating-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent (e or E) followed by a possibly signed integer value.	Pointer to FLOAT.
"c"	Sequence of bytes; white-space characters that are ordinarily skipped are read when "c" is specified.	Pointer to CHAR large enough for input field.
"lc, C"	Multibyte characters. The multibyte characters are converted to wide characters as if by a call to mbstowcs. The field width specifies the number of wide characters matched; if no width is specified, one multibyte character is matched.	Pointer to wchar_t large enough for input field.
"s"	Sequence of bytes that do not include white space.	Pointer to char array large enough for input field plus a terminating null character ("\0"), which is automatically appended.
"ls, S"	Multibyte string. None of the characters can be single-byte white-space characters (as specified by the isspace function). Each multibyte character in the sequence is converted to a wide character as if by a call to mbstowcs.	Pointer to wchar_t array large enough for the input field and the terminating null wide character ("L\0"), which is added automatically.
"n"	No input read from stream or buffer.	Pointer to INT into which is stored the number of characters successfully read from the stream or buffer up to that point in the call to scanf.
"p"	Pointer to "void" converted to series of characters.	Pointer to "void".

To read strings not delimited by space characters, substitute a set of characters in brackets ([ ]) for the *s* (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification %*a*c, where *a* is a decimal integer. In this instance, the c type character means that the argument is a pointer to a character array. The next number of characters is read from the input stream into the specified location, and no null character is added.

The input for a %*x* format specifier is interpreted as a hexadecimal number.

`scanf` scans each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on stdin.

Returns

`scanf` returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is `EOF` for an attempt to read at end-of-file if no conversion was performed. A return value of `0` means that no fields were assigned.

Example Code

This example scans various types of data.

```
#include <stdio.h>

int main(void)
{
   int i;
   float fp;
   char c,s[81];

   printf("Enter an integer, a real number, a character and a string : \n");
   if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
      printf("Not all of the fields were assigned\n");
   else {
      printf("integer = %d\n", i);
      printf("real number = %f\n", fp);
      printf("character = %c\n", c);
      printf("string = %s\n", s);
   }
   return 0;

   /*****************************************************************************
      The output should be similar to:

      Enter an integer, a real number, a character and a string :
      12 2.5 a yes
      integer = 12
      real number = 2.500000
      character = a
      string = yes
   *****************************************************************************/
}
```

This example converts a hexadecimal integer to a decimal integer. The `while` loop ends if the input value is not a hexadecimal integer.

```
#include <stdio.h>

int main(void)
{
   int number;

   printf("Enter a hexadecimal number or anything else to quit:\n");
   while (scanf("%x", &number)) {
      printf("Hexadecimal Number = %x\n", number);
      printf("Decimal Number     = %d\n", number);
   }
   return 0;

   /*****************************************************************************
      The output should be similar to:

      Enter a hexadecimal number or anything else to quit:
      0x231
      Hexadecimal Number = 231
      Decimal Number     = 561
      0xf5e
      Hexadecimal Number = f5e
      Decimal Number     = 3934
      0x1
      Hexadecimal Number = 1
      Decimal Number     = 1
      q
   ****************************************************************************/
}
```

Related Information

- _cscanf
- fscanf
- printf
- sscanf

_searchenv - Search for File

_searchenv - Search for File

Syntax

```
#include <stdlib.h>
void _searchenv(char *name, char *env_var, char *path);
```

Description

_searchenv searches for the target file in the specified domain. The *env_var* variable can be any environment variable that specifies a list of directory paths, such as PATH, LIB, INCLUDE, or other user-defined variables. Most often, it is PATH, causing a search for *name* in all directories specified in the PATH variable.

The routine first searches for the file in the current working directory. If it does not find the file, it next looks through the directories specified by the environment variable.

If the target file is found in one of the directories, the fully-qualified file name is copied into the buffer that *path* points to. You must ensure sufficient space for the constructed file name. If the target file is not found, *path* contains an empty null-terminated string.

Returns

There is no return value.

Example Code

This example searches for the files `searchen.c` and `cmd.exe`.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char path_buffer[_MAX_PATH];

   _searchenv("cmd.exe", "PATH", path_buffer);
   printf("path: %s\n", path_buffer);
   _searchenv("_searche.c", "DPATH", path_buffer);
   printf("path: %s\n", path_buffer);
   return 0;

   /****************************************************************************
      The output should be similar to:

      path: C:\OS2\cmd.exe
      path: C:\src\_searche.c
   ****************************************************************************/
}
```

Related Information

- getenv
- putenv

----

setbuf - Control Buffering

setbuf - Control Buffering

Syntax

```
#include <stdio.h>
void setbuf(FILE *stream, char *buffer);
```

Description

`setbuf` controls buffering for the specified *stream* . The *stream* pointer must refer to an open file before any I/O or repositioning has been done.

If the *buffer* argument is `NULL`, the *stream* is unbuffered. If not, the *buffer* must point to a character array of length `BUFSIZ`, which is the buffer size defined in the `<stdio.h>` include file. The system uses the *buffer* , which you specify, for input/output buffering instead of the default system-allocated buffer for the given *stream* .

`setvbuf` is more flexible than `setbuf`.

Note: Streams are fully buffered by default, with the exceptions of stderr, which is line-buffered, and memory files, which are unbuffered.

Returns

There is no return value.

Example Code

This example opens the file `setbuf.dat` for writing. It then calls `setbuf` to establish a buffer of length `BUFSIZ`. When `string` is written to the stream, the buffer `buf` is used and contains the string before it is flushed to the file.

```
#include <stdio.h>

#define FILENAME  "setbuf.dat"

int main(void)
{
   char buf[BUFSIZ];
   char string[] = "hello world";
   FILE *stream;

   memset(buf, '\0', BUFSIZ);           /* initialize buf to null characters  */
   stream = fopen(FILENAME, "wb");
   setbuf(stream, buf);                                   /* set up buffer    */
   fwrite(string, sizeof(string), 1, stream);
   printf("%s\n", buf);                   /* string is found in buf now       */
   fclose(stream);                  /* buffer is flushed out to output stream */
   return 0;

   /****************************************************************************
```

```
      The output file should contain:

      hello world

      The output should be:

      hello world
   *****************************************************************************/
}
```

Related Information

- fclose
- fflush
- _flushall
- fopen
- setvbuf

_set_crt_msg_handle - Change Run-Time Message Output Destina

_set_crt_msg_handle - Change Run-Time Message Output Destination

Syntax

```
#include <stdio.h>
int _set_crt_msg_handle(int fh);
```

Description

_set_crt_msg_handle changes the file handle to which run-time messages are sent, which is usually file handle 2, to *fh* . Run-time messages include exception handling messages and output from debug memory management routines.

Use _set_crt_msg_handle to trap run-time message output in applications where handle 2 is not defined, such as Presentation Manager applications.

The file handle *fh* must be a writable file or pipe handle. Set *fh* only for the current library environment.

Returns

_set_crt_msg_handle returns the handle to be used for run-time message output. If an handle that is not valid is passed as an argument, it is ignored and no change takes place.

Example Code

This example causes an exception by dereferencing a null pointer and uses _set_crt_msg_handle to send the exception messages to the file `_scmhdl.out`.

```
#include <sys\stat.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int   fh;
   char  *p = NULL;

   if (-1 == (fh = open("_scmhdl.out", O_CREAT|O_TRUNC|O_RDWR,
                        S_IREAD|S_IWRITE))) {
      perror("Unable to open the file _scmhdl.out.");
      exit(EXIT_FAILURE);
   }
   /* change file handle where messages are sent */
   if (fh != _set_crt_msg_handle(fh)) {
      perror("Could not change massage output handle.");
      exit(EXIT_FAILURE);
   }
   *p = 'x';          /* cause an exception, output should be in _scmhdl.out */
   if (-1 == close(fh)) {
      perror("Unable to close _scmhdl.out.");
      exit(EXIT_FAILURE);
   }
   return 0;

   /*****************************************************************************
      Running this program would cause an exception to occur,
      the file _scmhdl.out should contain the exception messages similar to :

      Exception = c0000005 occurred at EIP = 10068.
   ****************************************************************************/
}
```

Related Information

- open
- fileno
- _sopen

----

setjmp - Preserve Environment

setjmp - Preserve Environment

Syntax

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

`setjmp` saves a stack environment that can subsequently be restored by `longjmp`. `setjmp` and `longjmp` provide a way to perform a nonlocal `goto`. They are often used in signal handlers.

A call to `setjmp` causes it to save the current stack environment in *env*. A subsequent call to `longjmp` restores the saved environment and returns control to a point corresponding to the `setjmp` call. The values of all variables (except register variables) accessible to the function receiving control contain the values they had when `longjmp` was called. The values of variables that are allocated to registers by the compiler are unpredictable. Because any `auto` variable could be allocated to a register in optimized code, you should consider the values of all `auto` variables to be unpredictable after a longjmp call.

C++ Considerations When you call setjmp in a C++ program, ensure that that part of the program does not also create C++ objects that need to be destroyed. When you call longjmp, objects existing at the time of the setjmp call will still exist, but any destructors called after setjmp are not called. See longjmp for an example.

Returns

`setjmp` returns the value `0` after saving the stack environment. If `setjmp` returns as a result of a `longjmp` call, it returns the *value* argument of `longjmp`, or `1` if the *value* argument of `longjmp` is `0`. There is no error return value.

Example Code

This example stores the stack environment at the statement

```
   if(setjmp(mark) != 0) ...
```

When the system first performs the `if` statement, it saves the environment in *mark* and sets the condition to `FALSE` because `setjmp` returns a 0 when it saves the environment. The program prints the message

```
   setjmp has been called
```

The subsequent call to function `p` tests for a local error condition, which can cause it to perform the `longjmp` function. Then, control returns to the original `setjmp` function using the environment saved in *mark* . This time, the condition is `TRUE` because `-1` is the return value from the `longjmp` function. The program then performs the statements in the block and prints

```
   longjmp has been called
```

Then the program calls the `recover` function and exits.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf mark;

void p(void)
{
   int error = 0;

   error = 9;

   if (error != 0)
      longjmp(mark, -1);
}

void recover(void)
{
}

int main(void)
{
   if (setjmp(mark) != 0) {
      printf("longjmp has been called\n");
      recover();
      return 0;
   }
   printf("setjmp has been called\n");

   p();

   return 0;

   /****************************************************************************
      The output should be:

      setjmp has been called
      longjmp has been called
   ****************************************************************************/
}
```

Related Information

- longjmp
- goto in the *Language Reference*

----

setlocale - Set the Locale of the Program

setlocale - Set the Locale of the Program

Syntax

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description

setlocale sets or queries the specified *category* of the program's *locale* . A *locale* is the complete definition of that part of a program that depends on language and cultural conventions.

The default locale is "C". You can accept the default, or you can use setlocale to set the locale to one of the supplied locales.

Note: The locale is global to all threads in a process, therefore a multithread process should not execute the setlocale function when other threads are executing for that process. The results are unpredictable if this occurs.

Some examples of the supplied locales as you would specify them for setlocale are:

`"En_GB.IBM-850"`	(English, Great Britain, code page 850)
`"Fr_CA.IBM-863"`	(French, France, code page 863)
`"Ja_JP.IBM-932"`	(Japanese, Japan, code page 932)

The result of setlocale depends on the arguments you specify:

- The most general way to use setlocale is to specify a null string (`""`) for *locale* , for example, setlocale (LC_ALL, ""). setlocale then checks locale-related environment variables in the program's environment to find a locale name or names to use for *category* . To set a *category* to a specific *locale* , specify both the category and locale names. For example:

  ```
     setlocale(LC_CTYPE, "fr_fr.IBM-850");
  ```

  sets the LC_CTYPE category according to the "fr_fr.IBM-850" locale. The category names and their purpose are described in the table below.

  If *locale* name is not specified (NULL string), setlocale gets the locale name from:

  1. LC_ALL, if it is defined and not null
  2. The environment variable with the same name as *category* , if it is defined and not null
  3. LANG, if it is defined and not null
  4. If none of these variables is defined to a valid locale name, setlocale uses the system default locale.

  For example, if the LC_ALL environment variable is set to "De_DE" (Germany locale):

  ```
     setlocale(LC_TIME, "");
  ```

  sets the LC_TIME category to the "De_DE" locale.

  If *category* is LC_ALL and *locale* is a null string, setlocale checks the environment variables in the same order listed above. However, it checks the variable for each *category* and sets the category to the locale specified, which could be different for each category. (By contrast, if you specified LC_ALL and a specific locale name, all categories would be set to the same locale that you specify.) The string returned lists all the locales for all the categories.

- To query the locale for a *category* , specify a null pointer for *locale* . setlocale then returns a string indicating the locale setting for that *category* , without changing it. For example:

  ```
     char *s = setlocale(LC_CTYPE, NULL);
  ```

  returns the current locale for the LC_CTYPE category.

You can set the *category* argument of setlocale to one of these values:

Category Purpose

LC_ALL Specifies all categories associated with the program's locale.

LC_COLLATE Affects the selection of the collation sequence; that is, the relative order of collation elements (characters and multicharacter collation elements) in the program's locale. The collation sequence definition is used by regular expression, pattern matching, and sorting functions. Affects the regular expression functions regcomp and regexec; the string functions strcoll, strxfrm, wcscoll, and wcsxfrm.

Because both LC_CTYPE and LC_COLLATE modify the same storage area, setting LC_CTYPE and LC_COLLATE to different categories causes unpredictable results.

LC_CTYPE Defines the selection of character classification and case conversion for characters in the program's locale. Affects the behavior of character-handling functions defined in the "<ctype.h>" header file: isalnum, isalpha, isblank, iswblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, iswalnum, iswalpha, iswcntrl, iswctype, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, isxdigit, tolower, toupper, towlower, towupper, and wctype.

Affects behavior of the printf and scanf families of functions: fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, fscanf, scanf, and sscanf.

Affects the behavior of wide character input/output functions: fgetwc, fgetws, getwc, getwchar, fputwc, fputws, putwc, putwchar, and ungetwc.

Affects the behavior of multibyte and wide character conversion functions: mblen, mbstowcs, mbtowc, wcstod, wcstol, wcstombs, wcstoul, wcswidth, wctomb, and wcwidth.

Because both LC_CTYPE and LC_COLLATE modify the same storage area, setting LC_CTYPE and LC_COLLATE to different categories causes unpredictable results.

LC_MESSAGES Affects the language of the messages. The setting is used by catopen, catgets, and catclose. Affects the values returned by nl_langinfo and also defines affirmative and negative response patterns.

LC_MONETARY Affects monetary information returned by localeconv and strfmon. It defines the rules and symbols used to format monetary numeric information in the program's locale. The formatting rules and symbols are strings. localeconv returns pointers to these strings with names found in the "<locale.h>" header file.

LC_NUMERIC Affects the decimal-point character for the formatted input/output and string conversion functions, and the nonmonetary formatting information returned by the localeconv function, specifically:

printf family of functions
scanf family of functions
strtod
atof.

LC_TIME — Affects time and date format information in the program's locale, for the strftime, strptime, and wcsftime functions.

Returns

setlocale returns a string that specifies the locale for the *category* . If you specified "" for *locale* , the string names the current locale that has been configured; otherwise, it indicates the new locale that the *category* was set to.

If you specified LC_ALL for *category* , the returned string can be either a single locale name or a list of the locale names for each category in the following order:

1. LC_COLLATE
2. LC_CTYPE
3. LC_NUMERIC
4. LC_MONETARY
5. LC_TIME
6. LC_MESSAGES

The string can be used on a subsequent call to restore that part of the program's locale. Because the returned string can be overwritten by subsequent calls to setlocale, you should copy the string if you plan to use it later.

If an error occurs, setlocale returns NULL and does not alter the program's locale. Errors can occur if the *category* or *locale* is not valid, or if the value of the environment variable for a category does not contain a valid locale.

Example Code

This example sets the locale of the program to be `"fr_fr.ibm-850"` and prints the string that is associated with the locale.

```
#include <stdio.h>
#include <locale.h>

int main(void)
{
   char *string;

   if (NULL == (string = setlocale(LC_ALL, "fr_fr.ibm-850")))
      printf("setlocale failed.\n");
   else
      printf("The current locale is set to %s.\n", string);
   return 0;

   /*****************************************************************************
      The output should be similar to :

      The current locale is set to fr_fr.ibm-850.
   *****************************************************************************/
}
```

This example uses `setenv` to set the value of the environment variable LC_TIME to FRAN, calls `setlocale` to set all categories to default values, then query all categories, and uses printf to print results.

```
#include <locale.h>
#include <stdio.h>

int main(void)
{
   char *string;
   if (NULL == (string = setlocale(LC_TIME, "fr_fr")))
      printf("setlocale failed.\n");
   else
      printf("The current locale categories are: \"%s\"\n", string);
   return 0;

   /****************************************************************************
      Assuming that the default setting is en_us, the output should be similar to :

      The current locale categories are: "en_us en_us en_us en_us fr_fr en_us"
   ****************************************************************************/
}
```

Related Information

- "Introduction to Locale" in the *Programming Guide*
- getenv
- localeconv

_setmode - Set File Translation Mode

_setmode - Set File Translation Mode

Syntax

```
#include <fcntl.h>
#include <io.h>
int _setmode(int handle, int mode);
```

Description

_setmode sets the translation mode of the file given by *handle* to *mode* . The *mode* must be one of the values in the following table: compact break=fit.

Value	Meaning
O_TEXT	Sets the translated text mode. Carriage-return line-feed combinations are translated into a single line feed on input. Line-feed characters are translated into carriage-return line-feed combinations on output.
O_BINARY	Sets the binary (untranslated) mode. The above translations are suppressed.

Use _setmode to change the translation mode of a file handle. The translation mode only affects the read and write functions. _setmode does not affect the translation mode of streams.

If a file handle is acquired other than by a call to open, creat, _sopen or fileno, you should call _setmode for that file handle before using it within the read or write functions.

Returns

_setmode returns the previous translation mode if successful. A return value of `-1` indicates an error, and `errno` is set to one of the following values: compact break=fit.

Value	Meaning
EBADF	The file handle is not a handle for an open file.
EINVAL	Incorrect *mode* (neither O_TEXT nor O_BINARY)

Example Code

This example uses open to create the file `setmode.dat` and writes to it. The program then uses _setmode to change the translation mode of `setmode.dat` from binary to text.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>

#define  FILENAME      "setmode.dat"

/* routine to validate return codes                                    */

void ckrc(int rc)
{
   if (-1 == rc) {
      printf("Unexpected return code = -1\n");
      remove(FILENAME);
      exit(EXIT_FAILURE);
   }
}

int main(void)
{
   int h;
   int xfer;
   int mode;
   char rbuf[256];
   char wbuf[] = "123\n456\n";

   ckrc(h = open(FILENAME, O_CREAT|O_RDWR|O_TRUNC|O_TEXT, S_IREAD|S_IWRITE));
   ckrc(write(h, wbuf, sizeof(wbuf)));     /* write the file (text)          */
   ckrc(lseek(h, 0, SEEK_SET));         /* seek back to the start of the file */
   ckrc(xfer = read(h, rbuf, 5));           /* read the file text             */
   printf("Read in %d characters (4 expected)\n", xfer);
   ckrc(mode = _setmode(h, O_BINARY));
   if (O_TEXT == mode)
      printf("Mode changed from binary to text\n");
   else
      printf("Previous mode was not text (unexpected)\n");
   ckrc(xfer = read(h, rbuf, 5));           /* read the file (binary)         */
   printf("Read in %d characters (5 expected)\n", xfer);
   ckrc(close(h));
   remove(FILENAME);
   return 0;

   /*****************************************************************************
      The output should be:

      Read in 4 characters (4 expected)
      Mode changed from binary to text
      Read in 5 characters (5 expected)
   *****************************************************************************/
}
```

Related Information

- creat
- open
- _sopen
- read
- write

----

setvbuf - Control Buffering

setvbuf - Control Buffering

Syntax

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Description

setvbuf allows control over the buffering strategy and buffer size for a specified stream. The stream must refer to a file that has been opened, but not read or written to. The array pointed to by *buf* designates an area that you provide that the C library may choose to use as a buffer for the stream. A *buf* value of NULL indicates that no such area is supplied and that the C library is to assume responsibility for managing its own buffers for the stream. If you supply a buffer, it must exist until the stream is closed.

The *type* must be one of the following: compact break=fit.

Value	**Meaning**
_IONBF	No buffer is used.
_IOFBF	Full buffering is used for input and output. Use *buf* as the buffer and *size* as the size of the buffer.
_IOLBF	Line buffering is used. The buffer is flushed when a new-line character is written, when the buffer is full, or when input is requested.

If *type* is _IOFBF or _IOLBF, *size* is the size of the supplied buffer. If *buf* is NULL, the C library takes *size* as the suggested size for its own buffer. If *type* is _IONBF, both *buf* and *size* are ignored.

The value for *size* must be greater than 0.

Returns

setvbuf returns 0 if successful. It returns nonzero if an invalid value was specified in the parameter list, or if the request cannot be performed.

Warning: The array used as the buffer must still exist when the specified *stream* is closed. For example, if the buffer is declared within the scope of a function block, the *stream* must be closed before the function is terminated and frees the storage allocated to the buffer.

Example Code

This example sets up a buffer of buf for stream1 and specifies that input to stream2 is to be unbuffered.

```
#include <stdio.h>
#include <stdlib.h>

#define  BUF_SIZE      1024
#define  FILE1         "setvbuf1.dat"
#define  FILE2         "setvbuf2.dat"

char buf[BUF_SIZE];
FILE *stream1,*stream2;

int main(void)
{
   int flag = EXIT_SUCCESS;

   stream1 = fopen(FILE1, "r");
   stream2 = fopen(FILE2, "r");

   /* stream1 uses a user-assigned buffer of BUF_SIZE bytes                   */

   if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0) {
      printf("Incorrect type or size of buffer\n");
      flag = EXIT_FAILURE;
   }

   /* stream2 is unbuffered                                                   */

   if (setvbuf(stream2, NULL, _IONBF, 0) != 0) {
      printf("Incorrect type or size of buffer\n");
      flag = EXIT_FAILURE;
   }
   fclose(stream1);
   fclose(stream2);
   return  flag;
}
```

Related Information

- fclose
- fflush
- _flushall
- fopen
- setbuf

----

signal - Handle Interrupt Signals

signal - Handle Interrupt Signals

Syntax

```
#include <signal.h>
void ( *signal(int sig, void (*sig_handler)(int)) )(int);
```

Description

signal function assigns the signal handler *sig_handler* to handle the interrupt signal *sig*. Signals can be reported as a result of a machine interrupt (for example, division by zero) or by an explicit request to report a signal by using the `raise` function.

The *sig* argument must be one of the signal constants defined in `<signal.h>`: compact break=fit.

Value	**Meaning**
SIGABRT	Abnormal termination signal sent by `abort`. Default action: end the program.
SIGBREAK	Ctrl-Break signal. Default action: end the program.
SIGFPE	Floating-point exceptions that are not masked, such as overflow, division by zero, and invalid operation. Default action: end the program and provide an error message. If machine-state dumps are enabled, a dump is also provided.
SIGILL	Instruction not allowed. Default action: end the program and provide an error message. If machine-state dumps are enabled, a dump is also provided.
SIGINT	Ctrl-C signal. Default action: end the program.
SIGSEGV	Access to memory not valid. Default action: end the program and provide an error message. If machine-state dumps are enabled, a dump is also provided.
SIGTERM	Program termination signal sent by the user. Default action: end the program.
SIGUSR1	Defined by the user. Default action: ignore the signal.
SIGUSR2	Defined by the user. Default action: ignore the signal.
SIGUSR3	Defined by the user. Default action: ignore the signal.

For *sig_handler*, you must specify either the SIG_DFL or SIG_IGN constant (also defined in `<signal.h>`), or the address of a function that takes an integer argument (the signal). The action taken when the interrupt signal is received depends on the value of *sig_handler*:

Value **Meaning**

SIG_DFL
Perform the default action. This is the initial setting for all signals. The default actions are described in the list of signals above. All files controlled by the process are closed, but buffers are not flushed.

SIG_IGN
Ignore the interrupt signal.

sig_handler
Call the function *sig_handler* , which you provide, to handle the signal specified.

Your signal handler function (*sig_handler*) must take two integer arguments. The first argument is always the signal identifier. The second argument is `0`, unless the signal is SIG_FPE. For SIG_FPE signals, the second argument passed is a floating-point error signal as defined in `<float.h>`. If your *sig_handler* returns, the calling process resumes running immediately following the point at which it received the interrupt signal.

After a signal is reported and the *sig_handler* is called, signal handling for that signal is reset to the default. Depending on the purpose of the signal handler, you may want to call signal inside *sig_handler* to reestablish *sig_handler* as the signal handler. You can also reset the default handling at any time by calling signal and specifying SIG_DFL.

Signals and signal handlers are not shared between threads. If you do not establish a handler for a specific signal within a thread, the default signal handling is used regardless of what handlers you may have established in other concurrent threads.

Note: If an exception occurs in a math or critical library function, it is handled by *The Developer's Toolkit* exception handler. Your *sig_handler* will **not** be called. For more information about signals and exceptions, refer to "Signal and OS/2 Exception Handling" in the *Programming Guide* .

Returns

All calls to signal return the address of the previous handler for the re-assigned signal.

A return value of SIG_ERR (defined in `<signal.h>`) indicates an error, and `errno` is set to EINVAL. The possible causes of the error are an incorrect *sig* value or an undefined value for *sig_handler* .

Example Code

In the following example, the call to signal in `main` establishes the function `handler` to process the interrupt signal raised by `abort`. The `handler` prints a message and returns to the system.

```
#define INCL_DOSFILEMGR
#include <os2.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void handler(int sig)
{
   UCHAR FileData[100];
   ULONG Wrote;

   strcpy(FileData, "Signal occurred.\n\r");
   DosWrite(2, (PVOID)FileData, strlen(FileData), &Wrote);
}

int main(void)
{
   if (SIG_ERR == signal(SIGABRT, handler)) {
      perror("Could not set SIGABRT");
      return EXIT_FAILURE;
   }
   abort();                             /* signal raised by abort             */
   return 0;                            /* code should not reach here         */

   /****************************************************************************
      The output should be:

      Signal occurred.
   ****************************************************************************/
}
```

Related Information

- abort

- atexit
- exit
- _exit
- raise
- _spawnl - _spawnvpe
- "Signal and OS/2 Exception Handling" in the *Programming Guide*

----

sin - Calculate Sine

sin - Calculate Sine

Syntax

```
#include <math.h>
double sin(double x);
```

Description

sin calculates the sine of *x*, with *x* expressed in radians. If *x* is too large, a partial loss of significance in the result may occur.

Returns

sin returns the value of the sine of *x*.

Example Code

This example computes *y* as the sine of pi/2.

```
#include <math.h>

int main(void)
{
   double pi,x,y;

   pi = 3.1415926535;
   x = pi/2;
   y = sin(x);
   printf("sin( %lf ) = %lf\n", x, y);
   return 0;

   /****************************************************************************
      The output should be:

      sin( 1.570796 ) = 1.000000
   ****************************************************************************/
}
```

Related Information

- asin
- cos
- sinh
- tan

----

sinh - Calculate Hyperbolic Sine

sinh - Calculate Hyperbolic Sine

Syntax

```
#include <math.h>
double sinh(double x);
```

Description

`sinh` calculates the hyperbolic sine of *x*, with *x* expressed in radians.

Returns

`sinh` returns the value of the hyperbolic sine of *x*. If the result is too large, `sinh` sets `errno` to `ERANGE` and returns the value `HUGE_VAL` (positive or negative, depending on the value of *x*).

Example Code

This example computes *y* as the hyperbolic sine of pi/2.

```
#include <math.h>

int main(void)
{
   double pi,x,y;

   pi = 3.1415926535;
   x = pi/2;
   y = sinh(x);
   printf("sinh( %lf ) = %lf\n", x, y);
   return 0;

   /****************************************************************************
      The output should be:

      sinh( 1.570796 ) = 2.301299
   ****************************************************************************/
}
```

Related Information

- asin
- cosh
- sin
- tanh

_sopen - Open Shared File

_sopen - Open Shared File

Syntax

```
#include <fcntl.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>
```

```
int _sopen(char *pathname, int oflag, int shflag, int pmode);
```

Description

_sopen opens the file specified by *pathname* and prepares the file for subsequent shared reading or writing as defined by *oflag* and *shflag*. The *oflag* is an integer expression formed by combining one or more of the constants defined in `<fcntl.h>`. When more than one constant is given, the constants are joined with the OR operator (|). compact break=fit.

Oflag	**Meaning**
O_APPEND	Reposition the file pointer to the end of the file before every write operation.
O_CREAT	Create and open a new file. This flag has no effect if the file specified by *pathname* exists.
O_EXCL	Return an error value if the file specified by *pathname* exists. This flag applies only when used with O_CREAT.
O_RDONLY	Open the file for reading only. If this flag is given, neither O_RDWR nor O_WRONLY can be given.
O_RDWR	Open the file for both reading and writing. If this flag is given, neither O_RDONLY nor O_WRONLY can be given.
O_TRUNC	Open and truncate an existing file to `0` length. The file must have write permission. The contents of the file are destroyed. On the OS/2 operating system, do not specify O_TRUNC with O_RDONLY.
O_WRONLY	Open the file for writing only. If this flag is given, neither O_RDONLY nor O_RDWR can be given.
O_BINARY	Open the file in binary (untranslated) mode.
O_TEXT	Open the file in text (translated) mode. (See "Stream Processing" in the *Programming Guide* for a description of text and binary mode.)

The *shflag* argument is one of the following constants, defined in `<share.h>`: compact break=fit.

Shflag	**Meaning**
SH_DENYRW	Deny read and write access to file.
SH_DENYWR	Deny write access to file.
SH_DENYRD	Deny read access to file.
SH_DENYNO	Permit read and write access.

There is no default value for the *shflag*.

The *pmode* argument is required only when you specify O_CREAT. If the file does not exist, *pmode* specifies the permission settings of the file, which are set when the new file is closed for the first time. If the file exists, the value of *pmode* is ignored. The *pmode* must be one of the following values, defined in `<sys\stat.h>`: compact break=fit.

Value	**Meaning**
S_IWRITE	Writing permitted
S_IREAD	Reading permitted
S_IREAD | S_IWRITE	Reading and writing permitted.

If write permission is not given, the file is read-only. On the OS/2 operating system, all files are readable; you cannot give write-only permission. Thus, the modes S_IWRITE and S_IREAD | S_IWRITE are equivalent. Specifying a *pmode* of S_IREAD is similar to making a file read-only with the ATTRIB system command.

_sopen applies the current file permission mask to *pmode* before setting the permissions. (See umask for information on file permission masks.)

Returns

_sopen returns a file handle for the opened file. A return value of -1 indicates an error, and errno is set to one of the following values: compact break=fit.

Value	Meaning
EACCESS	The given path name is a directory, but the file is read-only and at attempt was made to open if for writing, or a sharing violation occurred.
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EMFILE	No more file handles are available.
ENOENT	The file or path name was not found.
EINVAL	An incorrect argument was passed.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example opens the file sopen.dat for shared reading and writing using _sopen. It then opens the file for shared reading.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <share.h>

#define FILENAME "sopen.dat"

int main(void)
{
   int fh1,fh2;

   printf("Creating file.\n");
   system("echo Sample Program > " FILENAME);

   /* share open the file for reading and writing                            */
   if (-1 == (fh1 = _sopen(FILENAME, O_RDWR, SH_DENYNO))) {
      perror("sopen failed");
      remove(FILENAME);
      return EXIT_FAILURE;
   }
   /* share open the file for reading only                                   */
   if (-1 == (fh2 = _sopen(FILENAME, O_RDONLY, SH_DENYNO))) {
      perror("sopen failed");
      close(fh1);
      remove(FILENAME);
      return EXIT_FAILURE;
   }
   printf("File successfully opened for sharing.\n");
   close(fh1);
   close(fh2);
   remove(FILENAME);
   return 0;

   /****************************************************************************
      The output should be:

      Creating file.
      File successfully opened for sharing.
   ****************************************************************************/
}
```

Related Information

- close
- creat
- open
- fdopen
- fopen
- _sopen
- umask

_spawnl - _spawnvpe - Start and Run Child Processes

_spawnl - _spawnvpe - Start and Run Child Processes

Syntax

```
#include <process.h>
int _spawnl(int modeflag, char *pathname, char *arg0, char *arg1, ...,
              char *argn, NULL);
int _spawnlp(int modeflag, char *pathname, char *arg0, char *arg1, ...,
              char *argn, NULL);
int _spawnle(int modeflag, char *pathname, char *arg0, char *arg1, ...,
              char *argn, NULL, char *envp[ ]);
int _spawnlpe(int modeflag, char *pathname, char *arg0, char *arg1, ...,
              char *argn, NULL, char *envp[ ]);
int _spawnv(int modeflag, char *pathname, char *argv[ ]);
int _spawnvp(int modeflag, char *pathname, char *argv[ ]);
int _spawnve(int modeflag, char *pathname, char *argv[ ], char *envp[ ]);
int _spawnvpe(int modeflag, char *pathname, char *argv[ ], char *envp[ ])
```

Description

Each of the _spawn functions creates and runs a new child process. Enough storage must be available for loading and running the child process. All of the _spawn functions are versions of the same routine; the letters at the end determine the specific variation: compact break=fit.

Letter	**Variation**
p	Uses PATH environment variable to find the file to be run
l	Lists command-line arguments separately
v	Passes to the child process an array of pointers to command-line arguments
e	Passes to the child process an array of pointers to environment strings.

The *modeflag* argument determines the action taken by the parent process before and during the _spawn. The values for *modeflag* are defined in `<process.h>`:   compact break=fit.

Value	**Meaning**
P_WAIT	Suspend the parent process until the child process is complete.
P_NOWAIT	Continue to run the parent process concurrently.
P_OVERLAY	Start the child process, and then, if successful, end the parent process. (This has the same effect as exec calls.)

The *pathname* argument specifies the file to run as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *pathname* does not have a file-name extension or end with a period, the _spawn functions add the extension .EXE and search for the file. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the _spawn functions search for *pathname* with no extension. The _spawnlp, _spawnlpe, _spawnvp, and _spawnvpe functions search for *pathname* (using the same procedures) in the directories specified by the PATH environment variable.

You pass arguments to the child process by giving one or more pointers to character strings as arguments in the _spawn routine. These character strings form the argument list for the child process.

The argument pointers can be passed as separate arguments (_spawnl, _spawnle, _spawnlp, and _spawnlpe) or as an array of pointers (_spawnv, _spawnve, _spawnvp, and _spawnvpe). At least one argument, either `arg0` or `argv[0]`, must be passed to the child process. By convention, this argument is a copy of the *pathname* argument. However, a different value will not produce an error.

Use the _spawnl, _spawnle, _spawnlp, and _spawnlpe functions where you know the number of arguments. The `arg0` is usually a pointer to *pathname*. The `arg1` through `arg`*n* arguments are pointers to the character strings forming the new argument list. Following `arg`*n*, a `NULL` pointer must mark the end of the argument list.

The _spawnv, _spawnve, _spawnvp, and _spawnvpe functions are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, argv. The argv[0] argument is usually a pointer to the *pathname* . The argv[1] through argv[*n*] arguments are pointers to the character strings forming the new argument list. The argv[*n*+1] argument must be a NULL pointer to mark the end of the argument list.

Files that are open when a _spawn call is made remain open in the child process. In the _spawnl, _spawnlp, _spawnv, and _spawnvp calls, the child process inherits the environment of the parent. The _spawnle, _spawnlpe, _spawnve, and _spawnvpe functions let you alter the environment for the child process by passing a list of environment settings through the envp argument. The envp argument is an array of character pointers, each element of which points to a null-terminated string, that defines an environment variable. Such a string has the form:

```
   NAME=value
```

where *NAME* is the name of an environment variable, and *value* is the string value to which that variable is set. (Notice that *value* is not enclosed in double quotation marks.) The final element of the envp array should be NULL. When envp itself is NULL, the child process inherits the environment settings of the parent process.

Note: Signal settings are not preserved in child processes created by calls to _spawn functions. The signal settings are reset to the default in the child process.

Returns

The return from a spawn function has one of two different meanings. The return value of a synchronous spawn is the exit status of the child process. The return value of an asynchronous spawn is the process identification of the child process. You can use wait or _cwait to get the child process exit code if an asynchronous spawn was done.

A return value of -1 indicates an error (the child process is not started), and errno is set to one of the following values: compact break=fit.

Value	Meaning
EAGAIN	The limit of the number of processes that the operating system permits has been reached.
EINVAL	The *modeflag* argument is incorrect.
ENOENT	The file or path name was not found or was not specified correctly.
ENOEXEC	The specified file is not executable or has an incorrect executable file format.
ENOMEM	Not enough storage is available to run the child process.

Example Code

This example calls four of the eight _spawn routines. When called without arguments from the command line, the program first runs the code for case PARENT. It spawns a copy of itself, waits for its child process to run, and then spawns a second child process. The instructions for the child process are blocked to run only if argv[0] and one parameter were passed (case CHILD). In its turn, each child process spawns a grandchild as a copy of the same program. The grandchild instructions are blocked by the existence of two passed parameters. The grandchild process can overlay the child process. Each of the processes prints a message identifying itself.

```
#include <stdio.h>
#include <process.h>

#define  PARENT        1
#define  CHILD         2
#define  GRANDCHILD    3

int main(int argc,char **argv,char **envp)
{
   int result;
   char *args[4];

   switch (argc) {
      case  PARENT :  /* no argument was passed: spawn child and wait         */
         result = _spawnle(P_WAIT, argv[0], argv[0], "one", NULL, envp);
         if (result)
            abort();
         args[0] = argv[0];
         args[1] = "two";
         args[2] = NULL;
```

```
          /* spawn another child, and wait for it                             */

         result = _spawnve(P_WAIT, argv[0], args, envp);
         if (result)
            abort();
         printf("Parent process ended\n");
         exit(0);



      case  CHILD :  /* one argument passed: allow grandchild to overlay      */
         printf("child process %s began\n", argv[1]);
         if ('o' == *argv[1])                             /* child one?       */
            {
            _spawnl(P_OVERLAY, argv[0], argv[0], "one", "two", NULL);
            abort();       /* not executed because child was overlaid         */
         }
         if ('t' == *argv[1])                             /* child two?       */
            {
            args[0] = argv[0];
            args[1] = "two";
            args[2] = "one";
            args[3] = NULL;
            _spawnv(P_OVERLAY, argv[0], args);
            abort();       /* not executed because child was overlaid         */
         }
         abort();                            /* argument not valid            */
      case  GRANDCHILD :   /* two arguments passed                            */
         printf("grandchild %s ran\n", argv[1]);
         exit(0);
   }
   /****************************************************************************
      The output should be:

      child process one began
      grandchild one ran
      child process two began
      grandchild two ran
      Parent process ended
   ****************************************************************************/
}
```

Related Information

- abort
- _cwait
- execl - _execvpe
- exit
- _exit
- wait

_splitpath - Decompose Path Name

_splitpath - Decompose Path Name

Syntax

```
#include <stdlib.h>
void _splitpath(char *path, char *drive, char *dir,
                    char *fname, char *ext);
```

Description

_splitpath decomposes an existing path name *path* into its four components. The *path* should point to a buffer containing the complete path name.

The maximum size necessary for each buffer is specified by the _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT constants defined in `<stdlib.h>`. The other arguments point to the following buffers used to store the path name elements: compact break=fit.

Buffer	Description
drive	Contains the drive letter followed by a colon (:) if a drive is specified in *path* .
dir	Contains the path of subdirectories, if any, including the trailing slash. Slashes (/), backslashes (\), or both may be present in *path* .
fname	Contains the base file name without any extensions.
ext	Contains the file name extension, if any, including the leading period (.).

You can specify `NULL` for any of the buffer pointers to indicate that you do not want the string for that component returned.

The return parameters contain empty strings for any path name components not found in *path* .

Returns

There is no return value.

Example Code

This example builds a file name path from the specified components, and then extracts the individual components.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath(path_buffer, "c", "qc\\bob\\eclibref\\e", "makepath", "c");
   printf("Path created with _makepath: %s\n\n", path_buffer);
   _splitpath(path_buffer, drive, dir, fname, ext);
   printf("Path extracted with _splitpath:\n");
   printf("drive: %s\n", drive);
   printf("directory: %s\n", dir);
   printf("file name: %s\n", fname);
   printf("extension: %s\n", ext);
   return 0;

   /***************************************************************************
      The output should be:

      Path created with _makepath: c:qc\bob\eclibref\e\makepath.c

      Path extracted with _splitpath:
      drive: c:
      directory: qc\bob\eclibref\e\
      file name: makepath
      extension: .c
   ***************************************************************************/
}
```

Related Information

- _fullpath
- _getcwd
- _getdcwd
- _makepath

----

sprintf - Print Formatted Data to Buffer

sprintf - Print Formatted Data to Buffer

Syntax

```
#include <stdio.h>
int sprintf(char *buffer, const char *format-string, argument-list);
```

Description

`sprintf` formats and stores a series of characters and values in the array *buffer* . Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string* .

The *format-string* consists of ordinary characters and has the same form and function as the *format-string* argument for the `printf` function. See printf for a description of the *format-string* and arguments.

In extended mode, `sprintf` also converts floating-point values of NaN and infinity to the strings `"NAN"` or `"nan"` and `"INFINITY"` or `"infinity"`. The case and sign of the string is determined by the format specifiers. See Infinity and NaN Support for more information on infinity and NaN values.

Returns

`sprintf` returns the number of bytes written in the array, not counting the ending null character.

Example Code

This example uses `sprintf` to format and print various data.

```
#include <stdio.h>

char buffer[200];
int i,j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
   c = 'l';
   i = 35;
   fp = 1.7320508;

    /* Format and print various data                                   */

   j = sprintf(buffer, "%s\n", s);
   j += sprintf(buffer+j, "%c\n", c);
   j += sprintf(buffer+j, "%d\n", i);
   j += sprintf(buffer+j, "%f\n", fp);
   printf("string:\n%s\ncharacter count = %d\n", buffer, j);
   return 0;

   /****************************************************************************
      The output should be:

      string:
      baltimore
      l
      35
      1.732051

      character count = 24
   ****************************************************************************/
}
```

Related Information

- Infinity and NaN Support

- _cprintf
- fprintf
- printf
- sscanf
- vfprintf
- vprintf
- vsprintf

----

sqrt - Calculate Square Root

sqrt - Calculate Square Root

Syntax

```
#include <math.h>
double sqrt(double x);
```

Description

sqrt calculates the nonnegative value of the square root of *x*.

Returns

sqrt returns the square root result. If *x* is negative, the function sets errno to EDOM, and returns 0.

Example Code

This example computes the square root of 45.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv)
{
   double value = 45.0;

   printf("sqrt( %f ) = %f\n", value, sqrt(value));
   return 0;

   /****************************************************************************
      The output should be:

      sqrt( 45.000000 ) = 6.708204
   ****************************************************************************/
}
```

Related Information

- exp
- hypot
- log
- log10
- pow

----

srand - Set Seed for rand Function

srand - Set Seed for rand Function

Syntax

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

`srand` sets the starting point for producing a series of pseudo-random integers. If `srand` is not called, the `rand` seed is set as if `srand(1)` were called at program start. Any other value for *seed* sets the generator to a different starting point.

The `rand` function generates the pseudo-random numbers.

Returns

There is no return value.

Example Code

This example first calls `srand` with a value other than 1 to initiate the random value sequence. Then the program computes five random values for the array of integers called `ranvals`.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int i,ranvals[5];

   srand(17);
   for (i = 0; i < 5; i++) {
      ranvals[i] = rand();
      printf("Iteration %d ranvals [%d] = %d\n", i+1, i, ranvals[i]);
   }
   return 0;

   /****************************************************************************
      The output should be similar to:

      Iteration 1 ranvals [0] = 24107
      Iteration 2 ranvals [1] = 16552
      Iteration 3 ranvals [2] = 12125
      Iteration 4 ranvals [3] = 9427
      Iteration 5 ranvals [4] = 13152
   ****************************************************************************/
}
```

Related Information

- rand

----

sscanf - Read Data

sscanf - Read Data

Syntax

```
#include <stdio.h>
int sscanf(const char *buffer, const char *format, argument-list);
```

Description

`sscanf` reads data from *buffer* into the locations given by *argument-list* . Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string* . See scanf for a description of the *format-string* .

Returns

`sscanf` returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is `EOF` when the end of the string is encountered before anything is converted.

Example Code

This example uses `sscanf` to read various data from the string *tokenstring* , and then displays that data.

```
#include <stdio.h>

#define  SIZE          81

char *tokenstring = "15 12 14";
int i;
float fp;
char s[SIZE];
char c;

int main(void)
{
    /* Input various data                                                     */

   sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp);

    /* If there were no space between %s and %c,                              */
    /* sscanf would read the first character following                        */
    /* the string, which is a blank space.                                    */
    /* Display the data                                                       */

   printf("string = %s\n", s);
   printf("character = %c\n", c);
   printf("integer = %d\n", i);
   printf("floating-point number = %f\n", fp);
   return 0;

   /*****************************************************************************
      The output should be:

      string = 15
      character = 1
      integer = 2
      floating-point number = 14.000000
   *****************************************************************************/
}

/*****************  Output should be similar to:  *****************

string = 15
character = 1
integer = 2
floating-point number = 14.000000
*/
```

Related Information

- Infinity and NaN Support

- _cscanf
- fscanf
- scanf
- sprintf

----

stat - Get Information about File or Directory

stat - Get Information about File or Directory

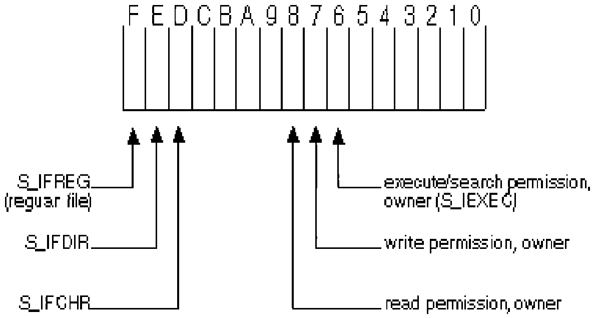
Syntax

```
#include <sys\types.h>
#include <sys\stat.h>
int stat(const char *pathname, struct stat *buffer)    ;
```

Description

stat stores information about the file or directory specified by *pathname* in the structure to which *buffer* points. The `stat` structure, defined in `<sys\stat.h>`, contains the following fields:

Field	**Value**
`st_mode`	Bit mask for file-mode information. stat sets the S_IFCHR bit if *handle* refers to a device. The S_IFDIR bit is set if *pathname* specifies a directory. The S_IFREG bit is set if *pathname* specifies an ordinary file. User read/write bits are set according to the permission mode of the file. The `S_IEXEC` bit is set using the file name extension. The other bits are undefined.
`st_dev`	Drive number of the disk containing the file.
`st_rdev`	Drive number of the disk containing the file (same as **st_dev**).
`st_nlink`	Always 1.
`st_size`	Size of the file in bytes.
`st_atime`	Time of last access of file.
`st_mtime`	Time of last modification of file.
`st_ctime`	Time of file creation.



Note: In earlier releases of C Set ++, stat began with an underscore (`_stat`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_stat` to stat for you.

Returns

stat returns the value `0` if the file status information is obtained. A return value of -1 indicates an error, and errno is set to ENOENT, indicating that the file name or path name could not be found.

Example Code

This example requests that the status information for the file `test.exe` be placed into the structure *buf*. If the request is successful and the file is executable, the example runs `test.exe`.

```
#include <sys\types.h>
#include <sys\stat.h>
#include <process.h>
#include <stdio.h>

int main(void)
{
   struct stat buf;

   if (0 == stat("test.exe", &buf)) {
      if ((buf.st_mode&S_IFREG) && (buf.st_mode&S_IEXEC))
         execl("test.exe", "test", NULL);         /* file is executable       */
   }
   else
      printf("File could not be found\n");
   return 0;

   /****************************************************************************
      The source for test.exe is:

      #include <stdio.h>
      int main(void)
      {
          puts("test.exe is an executable file");
      }

      The output should be:

      test.exe is an executable file
   ****************************************************************************/
}
```

Related Information

- fstat

----

strcat - Concatenate Strings

strcat - Concatenate Strings

Syntax

```
#include <string.h>
char *strcat(char *string1, const char *string2);
```

Description

`strcat` concatenates *string2* to *string1* and ends the resulting string with the null character.

`strcat` operates on null-terminated strings. The string arguments to the function should contain a null character (`\0`) marking the end of the string. No length checking is performed. You should not use a literal string for a *string1* value, although *string2* may be a literal string.

If the storage of *string1* overlaps the storage of *string2*, the behavior is undefined.

Returns

`strcat` returns a pointer to the concatenated string (*string1*).

Example Code

This example creates the string `"computer program"` using `strcat`.

```
#include <stdio.h>
#include <string.h>

#define  SIZE         40

int main(void)
{
   char buffer1[SIZE] = "computer";
   char *ptr;

   ptr = strcat(buffer1, " program");
   printf("buffer1 = %s\n", buffer1);
   return 0;

   /*****************************************************************************
      The output should be:

      buffer1 = computer program
   *****************************************************************************/
}
```

Related Information

- strchr
- strcmp
- strcpy
- strcspn
- strncat
- wcscat
- wcsncat

---

strchr - Search for Byte

strchr - Search for Byte

Syntax

```
#include <string.h>
char *strchr(const char *string, int c);
```

Description

`strchr` finds the first occurrence of a byte in a string. The byte *c* can be the null byte (`\0`); the ending null byte of *string* is included in the search.

The `strchr` function operates on null-terminated strings. The string arguments to the function should contain a null byte (`\0`) marking the end of the string.

Returns

`strchr` returns a pointer to the first occurrence of *c* in *string*. The function returns `NULL` if the specified *byte* is not found.

Example Code

This example finds the first occurrence of the character `p` in `"computer program"`.

```
#include <stdio.h>
#include <string.h>

#define  SIZE         40

int main(void)
{
   char buffer1[SIZE] = "computer program";
   char *ptr;
   int ch = 'p';

   ptr = strchr(buffer1, ch);
   printf("The first occurrence of %c in '%s' is '%s'\n", ch, buffer1, ptr);
   return 0;

   /****************************************************************************
      The output should be:

      The first occurrence of p in 'computer program' is 'puter program'
   ****************************************************************************/
}
```

Related Information

- strcmp
- strcspn
- strncmp
- strpbrk
- strrchr
- strspn
- wcschr
- wcsspn

----

strcmp - Compare Strings

strcmp - Compare Strings

Syntax

```
#include <string.h>
int strcmp(const char *string1, const char *string2);
```

Description

`strcmp` compares the strings pointed to by *string1* and *string2*. The function operates on null-terminated strings. The string arguments to the function should contain a null character (`\0`) marking the end of the string.

Returns

`strcmp` returns a value indicating the relationship between the two strings, as follows: compact break=fit.

Value	Meaning
Less than `0`	*string1* less than *string2*
`0`	*string1* identical to *string2*

Greater than 0	*string1* greater than *string2*.

Example Code

This example compares the two strings passed to `main` using `strcmp`.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   int result;

   if (argc != 3) {
      printf("Usage: %s string1 string2\n", argv[0]);
   }
   else {
      result = strcmp(argv[1], argv[2]);
      if (0 == result)
         printf("\"%s\" is identical to \"%s\"\n", argv[1], argv[2]);
      else
         if (result < 0)
            printf("\"%s\" is less than \"%s\"\n", argv[1], argv[2]);
         else
            printf("\"%s\" is greater than \"%s\"\n", argv[1], argv[2]);
   }
   return 0;

   /****************************************************************************
      If the following arguments are passed to this program:

      "is this first?" "is this before that one?"

      The output should be:

      "is this first?" is greater than "is this before that one?"
   ****************************************************************************/
}
```

Related Information

- strcmpi
- strcoll
- strcspn
- stricmp
- strncmp
- strnicmp
- wcscmp
- wcsncmp

----

strcmpi - Compare Strings Without Case Sensitivity

strcmpi - Compare Strings Without Case Sensitivity

Syntax

```
#include <string.h>
int strcmpi(const char *string1, const char *string2);
```

Description

strcmpi compares *string1* and *string2* without sensitivity to case. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

Returns

strcmpi returns a value indicating the relationship between the two strings, as follows: compact break=fit.

Value	Meaning
Less than 0	*string1* less than *string2*
0	*string1* equivalent to *string2*
Greater than 0	*string1* greater than *string2* .

Example Code

This example uses strcmpi to compare two strings.

```
#include <stdio.h>
#include <string.h>

int main(void)
{

   /* Compare two strings without regard to case                              */

   if (0 == strcmpi("hello", "HELLO"))
      printf("The strings are equivalent.\n");
   else
      printf("The strings are not equivalent.\n");
   return 0;

   /****************************************************************************
      The output should be:

      The strings are equivalent.
   ****************************************************************************/
}
```

Related Information

- strcoll
- strcspn
- strdup
- stricmp
- strncmp
- strnicmp
- wcscmp
- wcsncmp

----

strcoll - Compare Strings Using Collation Rules

strcoll - Compare Strings Using Collation Rules

Syntax

```
#include <string.h>
int strcoll(const char *string1, const char *string2);
```

Description

strcoll compares the string pointed to by *string1* against the string pointed to by *string2* , both interpreted according to the information in the LC_COLLATE category of the current locale.

strcoll differs from the strcmp function. strcoll performs a comparison between two character strings based on language collation rules as controlled by the LC_COLLATE category. In contrast, strcmp performs a character to character comparison.

- If a string will be collated many times, as when inserting an entry into a sorted list, strxfrm followed by strcmp can be more efficient.
- If only comparison for strict equality is desired, strcmp should be used, since it is more efficient.

Returns

strcoll returns an integer value indicating the relationship between the strings, as listed below: compact break=fit.

Value	Meaning
Less than 0	*string1* is less than *string2*
0	*string1* is equivalent to *string2*
Greater than 0	*string1* is greater than *string2*

Note:

- strcoll might need to allocate additional memory to perform the comparison algorithm specified in the LC_COLLATE. If the memory request cannot be satisfied (by malloc), then strcoll fails.

- If the locale supports double-byte characters (MB_CUR_MAX specified as 2), strcoll validates the multibyte characters. strcoll fails if the string contains invalid multibyte characters.

Related Information

This example compares the two strings passed to *main* .

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   int result;

   if (argc != 3) {
      printf("Usage: %s string1 string2\n", argv[0]);
   }
   else {
      result = strcoll(argv[1], argv[2]);
      if (0 == result)
         printf("\"%s\" is identical to \"%s\"\n", argv[1], argv[2]);
      else
         if (result < 0)
            printf("\"%s\" is less than \"%s\"\n", argv[1], argv[2]);
         else
            printf("\"%s\" is greater than \"%s\"\n", argv[1], argv[2]);
   }
   return 0;

   /****************************************************************************
      If the program is passed the following arguments:

      "firststring" "secondstring"

      The output should be:

      "firststring" is less than "secondstring"
   ****************************************************************************/
}
```

Related Information

- setlocale
- strcmp
- strcmpi
- strncmp
- wcscoll

----

strcpy - Copy Strings

strcpy - Copy Strings

Syntax

```
#include <string.h>
char *strcpy(char *string1, const char *string2);
```

Description

`strcpy` copies *string2*, including the ending null character, to the location specified by *string1*.

`strcpy` operates on null-terminated strings. The string arguments to the function should contain a null character (`\0`) marking the end of the string. No length checking is performed. You should not use a literal string for a *string1* value, although *string2* may be a literal string.

Returns

`strcpy` returns a pointer to the copied string (*string1*.).

Example Code

This example copies the contents of `source` to `destination`.

```
#include <stdio.h>
#include <string.h>

#define  SIZE          40

int main(void)
{
   char source[SIZE] = "123456789";
   char source1[SIZE] = "123456789";
   char destination[SIZE] = "abcdefg";
   char destination1[SIZE] = "abcdefg";
   char *return_string;
   int index = 5;

   /* This is how strcpy works                                                */

   printf("destination is originally = '%s'\n", destination);
   return_string = strcpy(destination, source);
   printf("After strcpy, destination becomes '%s'\n\n", destination);

   /* This is how strncpy works                                               */

   printf("destination1 is originally = '%s'\n", destination1);
   return_string = strncpy(destination1, source1, index);
   printf("After strncpy, destination1 becomes '%s'\n", destination1);
   return 0;

   /****************************************************************************
      The output should be:

      destination is originally = 'abcdefg'
      After strcpy, destination becomes '123456789'
      destination1 is originally = 'abcdefg'
      After strncpy, destination1 becomes '12345fg'
   ****************************************************************************/
}
```

Related Information

- strcat
- strdup
- strncpy
- wcscpy
- wcsncpy

----

strcspn - Get Length of a Substring

strcspn - Get Length of a Substring

Syntax

```
#include <string.h>
size_t strcspn(const char *string1, const char *string2);
```

Description

`strcspn` finds the first occurrence of a byte in *string1* that belongs to the set of bytes specified by *string2* and calculates the length of the substring pointed to by *string1* . Ending null characters are not considered in the search.

The `strcspn` function operates on null-terminated strings. The string arguments to the function should contain a null byte (`\0`) marking the end of the string.

Returns

`strcspn` returns the index of the first character found. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters not in *string2* .

Example Code

This example uses `strcspn` to find the first occurrence of any of the characters `a`, `x`, `l` or `e` in `string`.

```
#include <stdio.h>
#include <string.h>

#define  SIZE          40

int main(void)
{
   char string[SIZE] = "This is the source string";
   char *substring = "axle";

   printf("The first %i characters in the string \"%s\" are not in the "
      "string \"%s\" \n", strcspn(string, substring), string, substring);
   return 0;

   /****************************************************************************
      The output should be:

      The first 10 characters in the string "This is the source string" are not
      in the string "axle"
   ****************************************************************************/
}
```

Related Information

- strchr
- strcmp
- strcmpi
- stricmp
- strncmp
- strnicmp
- strpbrk
- strspn
- wcscmp
- wcsncmp

_strdate - Copy Current Date

_strdate - Copy Current Date

Syntax

```
#include <time.h>
char *_strdate(char *date);
```

Description

_strdate stores the current date as a string in the buffer pointed to by *date* in the following format:

```
   mm/dd/yy
```

The two digits *mm* represent the month, the digits *dd* represent the day of the month, and the digits *yy* represent the year. For example, the string 10/08/91 represents October 8, 1991. The buffer must be at least 9 bytes.

Note: The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Returns

_strdate returns a pointer to the buffer containing the date string. There is no error return.

Example Code

This example prints the current date.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   char buffer[9];

   printf("The current date is %s \n", _strdate(buffer));
   return 0;

   /****************************************************************************
      The output should be similar to:

      The current date is 01/02/95
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- _ftime
- gmtime
- localtime
- mktime
- time
- tzset

----

strdup - Duplicate String

strdup - Duplicate String

Syntax

```
#include <string.h>
char *strdup(const char *string);
```

Description

strdup reserves storage space for a copy of *string* by calling `malloc`. The string argument to this function is expected to contain a null character (`\0`) marking the end of the string. Remember to free the storage reserved with the call to `strdup`.

Returns

strdup returns a pointer to the storage space containing the copied string. If it cannot reserve storage strdup returns `NULL`.

Example Code

This example uses strdup to duplicate a string and print the copy.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string = "this is a copy";
   char *newstr;

   /* Make newstr point to a duplicate of string                              */

   if ((newstr = strdup(string)) != NULL)
      printf("The new string is: %s\n", newstr);
   return 0;

   /****************************************************************************
      The output should be:

      The new string is: this is a copy
   ****************************************************************************/
}
```

Related Information

- strcpy
- strncpy
- wcscpy
- wcsncpy

----

strerror - Set Pointer to Run-Time Error Message

strerror - Set Pointer to Run-Time Error Message

Syntax

```
#include <string.h>
char *strerror(int errnum);
```

Description

`strerror` maps the error number in *errnum* to an error message string.

Returns

`strerror` returns a pointer to the string. The contents of the error message string is determined by the setting of the LC_MESSAGES category in the current locale.

The value of `errno` may be set to:

EILSEQ	An encoding error has occurred converting a multibyte character.
E2BIG	The output buffer is too small.

Example Code

This example opens a file and prints a run-time error message if an error occurs.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define FILENAME "strerror.dat"

int main(void)
{
   FILE *stream;

   if (NULL == (stream = fopen(FILENAME, "r")))
      printf(" %s \n", strerror(errno));
   return 0;
}
```

Related Information

- clearerr
- ferror
- perror
- _strerror

_strerror - Set Pointer to System Error String

_strerror - Set Pointer to System Error String

Syntax

```
#include <string.h>
char *_strerror(char *string);
```

Description

_strerror tests for system error. It gets the system error message for the last library call that produced an error and prefaces it with your *string* message.

Your *string* message can be a maximum of 94 bytes.

Unlike perror, _strerror by itself does not print a message. To print the message returned by _strerror to stdout, use a `printf` statement similar to the following:

```
   if ((access("datafile",2)) == -1)
      printf(stderr,_strerror(NULL));
```

You could also print the message to stderr with an `fprintf` statement.

To produce accurate results, call _strerror immediately after a library function returns with an error. Otherwise, subsequent calls might write over the errno value.

Returns

If *string* is equal to `NULL`, _strerror returns a pointer to a string containing the system error message for the last library call that produced an error, ended by a new-line character (`\n`).

If *string* is not equal to `NULL`, _strerror returns a pointer to a string containing:

- Your string message
- A colon
- A space
- The system error message for the last library call producing an error
- A new-line character (`\n`).

Example Code

This example shows how `_strerror` can be used with the `fopen` function.

```
#include <string.h>
#include <stdio.h>

#define INFILE    "_strerro.in"
#define OUTFILE   "_strerro.out"

int main(void)
{
   FILE *fh1,*fh2;

   fh1 = fopen(INFILE, "r");
   if (NULL == fh1)
     /*  the error message goes through stdout not stderr                     */
      printf(_strerror("Open failed on input file"));
   fh2 = fopen(OUTFILE, "w+");
   if (NULL == fh2)
      printf(_strerror("Open failed on output file"));
   else
      printf("Open on output file was successful.\n");
   if (fh1 != NULL)
      fclose(fh1);
   if (fh2 != NULL)
      fclose(fh2);
   remove(OUTFILE);
   return 0;

   /*****************************************************************************
      The output should be:

      Open failed on input file: The file cannot be found.
      Open on output file was successful.
   *****************************************************************************/
}
```

Related Information

- clearerr
- ferror
- perror
- strerror

----

strfmon - Convert Monetary Value to String

strfmon - Convert Monetary Value to String

Syntax

```
#include <monetary.h>
int strfmon(char *s, size_t maxsize, const char *format, ...);
```

Description

strfmon places characters into the array pointed to by **s*, as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The character string *format* contains two types of objects:

- Plain characters, which are copied to the output array.
- Directives, each of which results in the fetching of zero or more arguments that are converted and formatted.

The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are simply ignored. If objects pointed to by *s* and *format* overlap, the behavior is undefined.

The directive (conversion specification) consists of the following sequence.

1. A % character
2. Optional flags, described below: =*f*, ^, then +, (, then !
3. Optional field width (may be preceded by -)
4. Optional left precision: #*n*
5. Optional right precision: .*p*
6. Required conversion character to indicate what conversion should be performed: i or n.

Each directive is replaced by the appropriate characters, as described in the following list:

%i The double argument is formatted according to the locale's international currency format (for example, in USA: USD 1,234.56).

%n The double argument is formatted according to the locale's national currency format (for example, in USA: $1,234.56).

%% is replaced by %. No argument is converted.

You can give optional conversion specifications immediately after the initial % of a directive in the following order:

Specifier	Meaning
=*f*	Specifies *f* as the numeric fill character. This flag is used in conjunction with the maximum digits specification #*n* (see below). The default numeric fill character is the space character. This option does not affect the other fill operations that always use a space as the fill character.
^	Formats the currency amount without thousands grouping characters. The default is to insert the grouping characters if defined for the current locale.
+ | (	Specifies the style of representing positive and negative currency amounts. You can specify only one of `+` or `(`. The `+` specifies to use the locale's equivalent of `+` and `-`. For example, in USA, the empty (null) string if positive and `-` if negative. The `(` specifies to use the locale's equivalent of enclosing negative amounts within parenthesis. If this option is not included, a default specified by the current locale is used.
!	Suppresses the currency symbol from the output conversion.
[-]*w*	A decimal digit string that specifies a minimum field width in which the result of the conversion is right-justified (or left-justified if the `-` flag is specified).
#*n*	A decimal digit string that specifies a maximum number of digits expected to be formatted to the left of the radix character. You can use this option to keep the formatted output from multiple calls to strfmon aligned in the same columns. You can also use it to fill unused positions with a special character, as in `$***123.45`. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more digit positions are required than specified, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character. (See the `=`*f* specification above). If thousands grouping is enabled, the behavior is: 1. Format the number as if it is an *n* digit number. 2. Insert fill characters to the left of the leftmost digit (for example, `$0001234.56` or `$***1234.56`). 3. Insert the separator character (for example, `$0,001,234.56` or `$*,**1,234.56`). 4. If the fill character is not the digit zero, the separators are replaced by the fill character (for example, `$****1,234.56`). To ensure alignment, any characters appearing before or after the number in the formatted output, such as currency or sign symbols, are padded with space characters to make their positive and negative formats an equal length.
.*p*	A decimal digit string that specifies the number of digits after the radix character. If the value of the precision *p* is `0`, no radix character appears. If this option is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

The LC_MONETARY category of the program's locale affects the behavior of this function, including the monetary radix character (which is different from the numeric radix character affected by the LC_NUMERIC category), the thousands (or alternate grouping) separator, the currency symbols, and formats.

Returns

If the total number of resulting bytes including the terminating null character is not more than *maxsize* , strfmon returns the number of bytes placed into the array pointed to by *s* , not including the terminating null character. Otherwise, strfmon returns `-1` and the contents of the array are indeterminate.

Example Code

This example uses strfmon to format the monetary value for `money`, then prints the resulting string.

```
#include <monetary.h>
#include <locale.h>
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void)
{
   char   string[100];     /* hold the string returned from strfmon() */
   double money = 1234.56;

   if (NULL == setlocale(LC_ALL, "en_us.ibm-850")) {
      printf("Unable to setlocale().\n");
      exit(EXIT_FAILURE);
   }
   strfmon(string, 100, "%i", money);
   printf("International currency format = \"%s\"\n", string);
   strfmon(string, 100, "%n", money);
   printf("National currency format      = \"%s\"\n", string);
   return 0;

   /***************************************************************************
      The output should be similar to :

      International currency format = "USD 1,234.56"
      National currency format      = "$1,234.56"
   ***************************************************************************/
}
```

Related Information

- strftime

---

strftime - Convert to Formatted Time

strftime - Convert to Formatted Time

Syntax

```
#include <time.h>
size_t strftime(char *dest, size_t maxsize,
                const char *format, const struct tm *timeptr);
```

Description

strftime converts the time and date specification in the *timeptr* structure into a character string. It then stores the null-terminated string in the array pointed to by *dest* according to the format string pointed to by *format* . *maxsize* specifies the maximum number of bytes that can be copied into the array.

The format string is a multibyte character string containing:

- Conversion specification characters, preceded by a % sign.
- Ordinary multibyte characters, which are copied into the array unchanged.

If data has the form of a conversion specifier, but is not one of the accepted specifiers, the characters following the % are copied to the output.

The characters that are converted are determined by the LC_TIME category of the current locale and by the values in the time structure pointed to by *timeptr* . The time structure pointed to by *timeptr* is usually obtained by calling the gmtime or localtime function.

When objects to be copied overlap, the behavior is undefined.

The following table lists the strftime conversion specifiers:

Specifier	Meaning
%a	Replace with abbreviated weekday name from locale.

%A Replace with full weekday name from locale.

%b Replace with abbreviated month name from locale.

%B Replace with full month name from locale.

%c Replace with date and time from locale.

%C Replace with locale's century number (year divided by 100 and truncated)

%d Replace with day of the month as a decimal number (01-31).

%D Insert date in mm/dd/yy form, regardless of locale.

%e Insert day of the month as a decimal number (" 1"-"31"). A space precedes a single digit.

%Ec Replace with the locale's alternative date and time representation.

%EC Replace with the name of the base year (period) in the locale's alternate representation.

%Ex Replace with the locale's alternative date representation.

%EX Replace with the locale's alternative time representation.

%Ey Replace with the offset from %EC (year only) in the locale's alternate representation.

%EY Replace with the full alternative year representation.

%h Replace with locale's abbreviated month name. This is the same as %b.

%H Replace with hour (24-hour clock) as a decimal number (00-23).

%I Replace with hour (12-hour clock) as a decimal number (01-12).

%j Replace with day of the year as a decimal number (001-366).

%m Replace with month as a decimal number (01-12).

%M Replace with minute as a decimal number (00-59).

%n Replace with a newline character.

%Od Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeroes if there is any alternative symbol for zero; otherwise fill with leading spaces.

%Oe Replace with the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.

%OH Replace with the hour (24-hour clock), using the locale's alternative numeric symbols.

%OI Replace with the hour (12-hour clock), using the locale's alternative numeric symbols.

%Om Replace with the month, using the locale's alternative numeric symbols.

%OM Replace with the minutes, using the locale's alternative numeric symbols.

%OS Replace with the seconds, using the locale's alternative numeric symbols.

%Ou Replace with the weekday as a decimal number (1 to 7), with 1 representing Monday, using the locale's alternative numeric symbols.

%OU Replace with the week number of the year (00-53), where Sunday is the first day of the week, using the locale's alternative numeric symbols.

&OV Replace with week number of the year (01-53), where Sunday is the first day of the week, using the locale's alternative numeric symbols.

%Ow Replace with the weekday (Sunday=0), using the locale's alternative numeric symbols.

% OW Replace with the week number of the year (01-53), where Monday is the first day of the week, using the locale's alternative numeric symbols.

%Oy Replace with the year (offset from %C) in the locale's alternative representation, using the locale's alternative numeric symbols.

%p Replace with the locale's equivalent of AM or PM.

%r Replace with a string equivalent to %I:%M:%S %p.

%R Replace with time in 24 hour notation (%H:%M)

%S Replace with second as a decimal number (00-61).

%t Replace with a tab.

%T Replace with a string equivalent to %H:%M:%S.

%u Replace with the weekday as a decimal number (1 to 7), with 1 representing Monday.

%U Replace with week number of the year (00-53), where Sunday is the first day of the week.

%V Replace with week number of the year (01-53), where Monday is the first day of the week. If the week of January 1 is four or more days in the new year, it is week 1; otherwise, it is week 53 of the previous year and the following week is week 1.

%w Replace with weekday (0-6), where Sunday is 0.

%W Replace with week number of the year (00-53), where Monday is the first day of the week.

%x Replace with date representation of locale.

%X Replace with time representation of locale.

%y Replace with year without the century as a decimal number (00-99).

%Y Replace with year including the century as a decimal number.

%Z Replace with name of time zone, or no characters if time zone is not available.

%% Replace with %.

For %Z, the `tm_isdst` flag in the `tm` structure passed to strftime specifies whether the time zone is standard or Daylight Savings time.

If data has the form of a directive, but is not one of the above, the characters following the % are copied to the output.

Returns

If the total number of bytes in the resulting string including the terminating null byte does not exceed *maxsize* , strftime returns the number of bytes placed into *dest* , not including the terminating null byte; otherwise, strftime returns `0` and the content of the string is indeterminate.

Example Code

This example gets the time and date from localtime, calls strftime to format it, and prints the resulting string.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   char dest[70];
   int ch;
   time_t temp;
   struct tm *timeptr;

   temp = time(NULL);
   timeptr = localtime(&temp);
   ch = strftime(dest, sizeof(dest)-1, "Today is %A,"" %b %d. \n Time: %I:%M %p"
      , timeptr);
   printf("%d characters placed in string to make: \n \n %s", ch, dest);
   return 0;

   /****************************************************************************
      The output should be similar to:

      41 characters placed in string to make:
       Today is Monday, Sep 16.
       Time: 06:31 pm
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- localtime
- setlocale
- strptime
- time
- wcsftime

----

stricmp - Compare Strings as Lowercase

stricmp - Compare Strings as Lowercase

Syntax

```
#include <string.h>
int stricmp(const char *string1, const char *string2);
```

Description

`stricmp` compares *string1* and *string2* without sensitivity for case. All alphabetic characters in the arguments *string1* and *string2* are converted to lowercase before the comparison using locale information. `stricmp` operates on null-terminated strings.

Returns

stricmp returns a value that indicates the following relationship between the two strings : compact break=fit.

Value	Meaning
Less than 0	*string1* less than *string2*
0	*string1* identical to *string2*
Greater than 0	*string1* greater than *string2*.

Example Code

This example uses stricmp to compare two strings.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *str1 = "this is a string";
   char *str2 = "THIS IS A STRING";

   /* Compare two strings without regard to case                              */

   if (stricmp(str1, str2))
      printf("str1 is not the same as str2\n");
   else
      printf("str1 is the same as str2\n");
   return 0;

   /****************************************************************************
      The output should be:

      str1 is the same as str2
   ****************************************************************************/
}
```

Related Information

- strcmp
- strcmpi
- strcspn
- strncmp
- strnicmp

- wcscmp
- wcsncmp

----

strlen - Determine String Length

strlen - Determine String Length

Syntax

```
#include <string.h>
size_t strlen(const char *string);
```

Description

`strlen` determines the length of *string* excluding the terminating null byte.

Returns

`strlen` returns the length of *string* .

Example Code

This example determines the length of a string.

```
#include <stdio.h>
#include <string.h>

int main(int argc,char **argv)
{
   char *String = "How long is this string?";

   printf("Length of string \"%s\" is %i.\n", String, strlen(String));
   return 0;

   /****************************************************************************
      The output should be:

      Length of string "How long is this string?" is 24.
   ****************************************************************************/
}
```

Related Information

- mblen
- strrev
- wcslen

----

strlwr - Convert Uppercase to Lowercase

strlwr - Convert Uppercase to Lowercase

Syntax

```
#include <string.h>
char *strlwr(char *string);
```

Description

strlwr converts any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

Returns

strlwr returns a pointer to the converted *string* . There is no error return.

Example Code

This example makes a copy in all lowercase of the string `"General Assembly"`, and then prints the copy.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *string = "General Assembly";
   char *copy;

   copy = strlwr(strdup(string));
   printf("Expected result: general assembly\n");
   printf("strlwr returned: %s\n", copy);
   return 0;

   /****************************************************************************
      The output should be:

      Expected result: general assembly
      strlwr returned: general assembly
   ****************************************************************************/
}
```

Related Information

- strupr
- _toascii - _tolower - _toupper
- tolower - toupper
- towlower - towupper

----

strncat - Concatenate Strings

strncat - Concatenate Strings

Syntax

```
#include <string.h>
char *strncat(char *string1, const char *string2, size_t count);
```

Description

`strncat` appends the first *count* bytes of *string2* to *string1* and ends the resulting string with a null byte (`\0`). If *count* is greater than the length of *string2* , the length of *string2* is used in place of *count* .

The `strncat` function operates on null-terminated strings. The string argument to the function should contain a null byte (`\0`) marking the end of the string.

Returns

`strncat` returns a pointer to the joined string (*string1*).

Example Code

This example demonstrates the difference between `strcat` and `strncat`. `strcat` appends the entire second string to the first, whereas `strncat` appends only the specified number of bytes in the second string to the first.

```
#include <stdio.h>
#include <string.h>

#define  SIZE          40

int main(void)
{
   char buffer1[SIZE] = "computer";
   char *ptr;

   /* Call strcat with buffer1 and " program"                                 */

   ptr = strcat(buffer1, " program");
   printf("strcat : buffer1 = \"%s\"\n", buffer1);

   /* Reset buffer1 to contain just the string "computer" again               */

   memset(buffer1, '\0', sizeof(buffer1));
   ptr = strcpy(buffer1, "computer");

   /* Call strncat with buffer1 and " program"                                */

   ptr = strncat(buffer1, " program", 3);
   printf("strncat: buffer1 = \"%s\"\n", buffer1);
   return 0;

   /****************************************************************************
      The output should be:

      strcat : buffer1 = "computer program"
      strncat: buffer1 = "computer pr"
   ****************************************************************************/
}
```

Related Information

- strcat
- strnicmp
- wcscat
- wcsncat

----

strncmp - Compare Strings

strncmp - Compare Strings

Syntax

```
#include <string.h>
int strncmp(const char *string1, const char *string2, size_t count);
```

Description

`strncmp` compares the first *count* bytes of *string1* and *string2* .

Returns

`strncmp` returns a value indicating the relationship between the substrings, as follows: compact break=fit.

Value	**Meaning**
Less than `0`	*substring1* less than *substring2*
`0`	*substring1* equivalent to *substring2*
Greater than `0`	*substring1* greater than *substring2*

Example Code

This example demonstrates the difference between `strcmp` and `strncmp`.

```
#include <stdio.h>
#include <string.h>

#define  SIZE          10

int main(void)
{
   int result;
   int index = 3;
   char buffer1[SIZE] = "abcdefg";
   char buffer2[SIZE] = "abcfg";
   void print_result(int, char *, char *);

   result = strcmp(buffer1, buffer2);
   printf("Comparison of each character\n");
   printf("  strcmp: ");
   print_result(result, buffer1, buffer2);
   result = strncmp(buffer1, buffer2, index);
   printf("\nComparison of only the first %i characters\n", index);
   printf("  strncmp: ");
   print_result(result, buffer1, buffer2);
   return 0;
   /****************************************************************************
      The output should be:

      Comparison of each character
        strcmp: "abcdefg" is less than "abcfg"

      Comparison of only the first 3 characters
        strncmp: "abcdefg" is identical to "abcfg"
   ****************************************************************************/
}

void print_result(int res,char *p_buffer1,char *p_buffer2)
{
   if (0 == res)
      printf("\"%s\" is identical to \"%s\"\n", p_buffer1, p_buffer2);
   else
      if (res < 0)
         printf("\"%s\" is less than \"%s\"\n", p_buffer1, p_buffer2);
      else
         printf("\"%s\" is greater than \"%s\"\n", p_buffer1, p_buffer2);
}
```

Related Information

- strcmp
- strcmpi
- strcoll
- strcspn
- stricmp
- strnicmp
- wcscmp

- wcsncmp

----

strncpy - Copy Strings

strncpy - Copy Strings

Syntax

```
#include <string.h>
char *strncpy(char *string1, const char *string2, size_t count);
```

Description

`strncpy` copies *count* bytes of *string2* to *string1* . If *count* is less than or equal to the length of *string2* , a null byte (`\0`) is *not* appended to the copied string. If *count* is greater than the length of *string2* , the *string1* result is padded with null bytes (`\0`) up to length *count* .

Returns

`strncpy` returns a pointer to *string1* .

Example Code

This example demonstrates the difference between `strcpy` and `strncpy`.

```
#include <stdio.h>
#include <string.h>

#define  SIZE          40

int main(void)
{
   char source[SIZE] = "123456789";
   char source1[SIZE] = "123456789";
   char destination[SIZE] = "abcdefg";
   char destination1[SIZE] = "abcdefg";
   char *return_string;
   int index = 5;

   /* This is how strcpy works                                                */

   printf("destination is originally = '%s'\n", destination);
   return_string = strcpy(destination, source);
   printf("After strcpy, destination becomes '%s'\n\n", destination);

   /* This is how strncpy works                                               */

   printf("destination1 is originally = '%s'\n", destination1);
   return_string = strncpy(destination1, source1, index);
   printf("After strncpy, destination1 becomes '%s'\n", destination1);
   return 0;

   /*****************************************************************************
      The output should be:

      destination is originally = 'abcdefg'
      After strcpy, destination becomes '123456789'

      destination1 is originally = 'abcdefg'
      After strncpy, destination1 becomes '12345fg'
   *****************************************************************************/
}
```

Related Information

- strcpy
- strdup
- strnicmp

----

strnicmp - Compare Strings Without Case Sensitivity

strnicmp - Compare Strings Without Case Sensitivity

Syntax

```
#include <string.h>
int strnicmp(const char *string1, const char *string2, int n);
```

Description

strnicmp compares, at most, the first *n* characters of *string1* and *string2*. It operates on null-terminated strings.

strnicmp is case-insensitive; the uppercase and lowercase forms of a letter are considered equivalent. Conversion to lowercase uses locale information.

Returns

strnicmp returns a value indicating the relationship between the substrings, as listed below: compact break=fit.

Value	**Meaning**
Less than 0	*substring1* less than *substring2*
0	*substring1* equivalent to *substring2*
Greater than 0	*substring1* greater than *substring2*.

Example Code

This example uses strnicmp to compare two strings.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *str1 = "THIS IS THE FIRST STRING";
   char *str2 = "This is the second string";
   int numresult;

     /* Compare the first 11 characters of str1 and str2
        without regard to case                                                */

   numresult = strnicmp(str1, str2, 11);
   if (numresult < 0)
      printf("String 1 is less than string2.\n");
   else
      if (numresult > 0)
         printf("String 1 is greater than string2.\n");
      else
         printf("The two strings are equivalent.\n");
   return 0;

   /****************************************************************************
      The output should be:
```

```
      The two strings are equivalent.
   ****************************************************************************/
}
```

Related Information

- strcmp
- strcmpi
- stricmp
- strncmp
- wcscmp
- wcsncmp

----

strnset - strset - Set Bytes in String

strnset - strset - Set Bytes in String

Syntax

```
#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);
```

Description

strnset sets, at most, the first *n* bytes of *string* to *c* (converted to a `char`). If *n* is greater than the length of *string*, the length of *string* is used in place of *n*. strset sets all bytes of *string*, except the ending null character (\`0`), to *c* (converted to a `char`).

For both functions, the string must be initialized and must end with a null character (`\0`).

Returns

Both strset and strnset return a pointer to the altered *string*. There is no error return value.

Example Code

In this example, strnset sets not more than four bytes of a string to the byte 'x'. Then the strset function changes any non-null bytes of the string to the byte 'k'.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *str = "abcdefghi";

   printf("This is the string: %s\n", str);
   printf("This is the string after strnset: %s\n", strnset(str, 'x', 4));
   printf("This is the string after strset: %s\n", strset(str, 'k'));
   return 0;

   /****************************************************************************
      The output should be:

      This is the string: abcdefghi
      This is the string after strnset: xxxxefghi
      This is the string after strset: kkkkkkkkk
   ****************************************************************************/
}
```

Related Information

- strchr
- strpbrk
- wcschr
- wcspbrk

----

strpbrk - Find Bytes in String

strpbrk - Find Bytes in String

Syntax

```
#include <string.h>
char *strpbrk(const char *string1, const char *string2);
```

Description

`strpbrk` locates the first occurrence in the string pointed to by *string1* of any bytes from the string pointed to by *string2* .

Returns

`strpbrk` returns a pointer to the byte. If *string1* and *string2* have no bytes in common, a `NULL` pointer is returned.

Example Code

This example returns a pointer to the first occurrence in the array *string* of either `a` or `b`.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *result,*string = "A Blue Danube";
   char *chars = "ab";

   result = strpbrk(string, chars);
   printf("The first occurrence of any of the characters \"%s\" in "
      "\"%s\" is \"%s\"\n", chars, string, result);
   return 0;

   /****************************************************************************
      The output should be:

      The first occurrence of any of the characters "ab" in
      "A Blue Danube" is "anube"
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strrchr
- strspn
- wcschr
- wcscspn
- wcspbrk

- wcsrchr
- wcswcs

----

strptime - Convert to Formatted Date and Time

strptime - Convert to Formatted Date and Time

Syntax

```
#include <time.h>
char *strptime(const char *buf, const char *fmt, struct tm *tm);
```

Description

strptime uses the format specified by *fmt* to convert the character string pointed to by *buf* to values that are stored in the structure pointed to by *tm* .

The **fmt* is composed of zero or more directives. Each directive is composed of one of the following:

- One or more white-space characters (as specified by the isspace function)
- An ordinary character (neither % nor a white-space character)
- A conversion specifier.

Each conversion specifier consists of a % character followed by a conversion character that specifies the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifiers.

For a directive composed of white-space characters, strptime scans input up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

For a directive that is an ordinary character, strptime scans the next character from the buffer. If the scanned character differs from the one comprising the directive, the directive fails and the differing and subsequent characters remain unscanned.

For a series of directives composed of %n, %t, white-space characters, or any combination, strptime scans up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

For any other conversion specification, strptime scans characters until a character matching the next directive is scanned, or until no more characters can be scanned. It then compares these characters, excepting the one matching the next directive, to the locale values associated with the conversion specifier. If a match is found, strptime sets the appropriate `tm` structure members to values corresponding to the locale information. Case is ignored when items in *buf* are matched, such as month or weekday names. If no match is found, strptime fails and no more characters are scanned.

The following tables list the conversion specifiers for strptime.

Specifier	Meaning
%a	Day of week, using locale's abbreviated or full weekday name.
%A	Day of week, using locale's abbreviated or full weekday name.
%b	Month, using locale's abbreviated or full month name.
%B	Month, using locale's abbreviated or full month name.
%c	Date and time, using locale's date and time.
%C	Century number (year divided by 100 and truncated to an integer)

%d Day of the month (1-31; leading zeros permitted but not required).

%D Date as %m/%d/%y.

%e Day of the month (1-31; leading zeros permitted but not required).

%h Month, using locale's abbreviated or full month name.

%H Hour (0-23; leading zeros permitted but not required).

%I Hour (0-12; leading zeros permitted but not required).

%j Day number of the year (001-366).

%m Month number (1-12; leading zeros permitted but not required).

%M Minute (0-59; leading zeros permitted but not required).

%n Any white space.

%p Locale's equivalent of AM or PM.

%r Time as %I:%M:%S a.m. or %I:%M:%S p.m.

%R Time in 24 hour notation (%H%M)

%S Seconds (0-61; leading zeros permitted but not required).

%t Tab character.

%T Time as %H:%M:%S.

%U Week number of the year (0-53; where Sunday is the first day of the week; leading zeros permitted but not required).

%w Weekday (0-6; where Sunday is 0; leading zeros permitted but not required).

%W Week number of the year (0-53; where Monday is the first day of the week; leading zeros permitted but not required).

%x Date, using locale's date format.

%X Time, using locale's time format.

%y Year within century (0-99; leading zeros permitted but not required).

%Y Year, including century.

%Z Time zone name

%% Replace with %.

Some directives can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

Specifier	Meaning
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's alternative representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's alternative representation.
%EY	Replace with the full alternative year representation.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeroes if there is any alternative symbol for zero; otherwise, fill with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%OH	Replace with the hour (24-hour clock), using the locale's alternative numeric symbols.
%OI	Replace with the hour (12-hour clock), using the locale's alternative numeric symbols.
%Om	Replace with the month, using the locale's alternative numeric symbols.
%OM	Replace with the minutes, using the locale's alternative numeric symbols.
%OS	Replace with the seconds, using the locale's alternative numeric symbols.
%OU	Replace with the week number of the year (Sunday as the first day of the week, rules corresponding to %U), using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0), using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (Monday as the first day of the week), using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation, using the locale's alternative numeric symbols.

Returns

If successful, strptime returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

Example Code

This example uses strptime to convert a string to the structure `xmas`, then prints the contents of the structure.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   struct tm xmas;

   if (NULL == strptime("12/25/94 12:00:01", "%D %T", &xmas)) {
      printf("strptime() failed.\n");
      exit(EXIT_FAILURE);
   }
   printf("tm_sec  = %3d\n", xmas.tm_sec );
   printf("tm_min  = %3d\n", xmas.tm_min );
   printf("tm_hour = %3d\n", xmas.tm_hour);
   printf("tm_mday = %3d\n", xmas.tm_mday);
   printf("tm_mon  = %3d\n", xmas.tm_mon );
   printf("tm_year = %3d\n", xmas.tm_year);
   return 0;

   /****************************************************************************
      The output should be similar to :

      tm_sec  =   1
      tm_min  =   0
      tm_hour =  12
      tm_mday =  25
      tm_mon  =  11
      tm_year =  94
   ****************************************************************************/
}
```

Related Information

- strftime
- wcsftime

----

strrchr - Find Last Occurrence of Byte in String

strrchr - Find Last Occurrence of Byte in String

Syntax

```
#include <string.h>
char *strrchr(const char *string, int c);
```

Description

`strrchr` finds the last occurrence of *c* (converted to a byte) in *string*. The ending null byte is considered part of the *string*.

Returns

`strrchr` returns a pointer to the last occurrence of *c* in *string*. If the given byte is not found, a `NULL` pointer is returned.

Example Code

This example compares the use of `strchr` and `strrchr`. It searches the string for the first and last occurrence of `p` in the string.

```
#include <stdio.h>
#include <string.h>

#define  SIZE          40

int main(void)
{
   char buf[SIZE] = "computer program";
   char *ptr;
   int ch = 'p';

   /* This illustrates strchr                                                */

   ptr = strchr(buf, ch);
   printf("The first occurrence of %c in '%s' is '%s'\n", ch, buf, ptr);

   /* This illustrates strrchr                                               */

   ptr = strrchr(buf, ch);
   printf("The last occurrence of %c in '%s' is '%s'\n", ch, buf, ptr);
   return 0;

   /****************************************************************************
      The output should be:

      The first occurrence of p in 'computer program' is 'puter program'
      The last occurrence of p in 'computer program' is 'program'
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strspn
- wcschr
- wcspbrk
- wcsrchr
- wcswcs

----

strrev - Reverse String

strrev - Reverse String

Syntax

```
#include <string.h>
char *strrev(char *string);
```

Description

strrev reverses the order of the characters in the given *string* . The ending null character (`\0`) remains in place.

Returns

strrev returns a pointer to the altered *string* . There is no error return value.

Example Code

This example determines whether a string is a *palindrome* . A palindrome is a string that reads the same forward and backward.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int palindrome(char *string)
{
   char *string2;
   int  rc;

   /* Duplicate string for comparison                                         */

   if (NULL == (string2 = strdup(string))) {
      printf("Storage could not be reserved for string\n");
      exit(EXIT_FAILURE);
   }

   /* If result equals 0, the string is a palindrome                          */

   rc = strcmp(string), strrev(string2));
   free(string2);
   return rc;
}

int main(void)
{
   char string[81];

   printf("Please enter a string.\n");
   scanf("%80s", string);
   if (palindrome(string))
      printf("The string is not a palindrome.\n");
   else
      printf("The string is a palindrome.\n");
   return 0;

   /****************************************************************************
      Sample output from program:

      Please enter a string.
      level
      The string is a palindrome.
      ... or ...
      Please enter a string.
      levels
      The string is not a palindrome.
   ****************************************************************************/
}
```

Related Information

- strcat
- strcmp
- strcpy
- strdup
- strnset - strset

----

strspn - Get Length of Substring

strspn - Get Length of Substring

Syntax

```
#include <string.h>
size_t strspn(const char *string1, const char *string2);
```

Description

strspn computes the number of bytes in the maximum initial segment of the string pointed to by *string1* , which consists entirely of bytes from the string pointed to by *string2* .

Returns

`strspn` returns the index of the first byte found. This value is equal to the length of the initial substring of *string1* that consists entirely of bytes from *string2* . If *string1* begins with a byte not in *string2* , `strspn` returns `0`. If all the bytes in *string1* are found in *string2* , the length of *string1* is returned.

Example Code

This example finds the first occurrence in the array *string* of a byte that is not an `a`, `b`, or `c`. Because the string in this example is `cabbage`, `strspn` returns `5`, the length of the segment of `cabbage` before a byte that is not an `a`, `b`, or `c`.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string = "cabbage";
   char *source = "abc";
   int index;

   index = strspn(string, "abc");
   printf("The first %d characters of \"%s\" are found in \"%s\"\n", index,
      string, source);
   return 0;

   /****************************************************************************
      The output should be:

      The first 5 characters of "cabbage" are found in "abc"
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- wcschr
- wcscspn
- wcspbrk
- wcsrchr
- wcsspn
- wcswcs

----

strstr - Locate Substring

strstr - Locate Substring

Syntax

```
#include <string.h>
char *strstr(const char *string1, const char *string2);
```

Description

`strstr` finds the first occurrence of *string2* in *string1* . The function ignores the null byte (`\0`) that ends *string2* in the matching process.

Returns

`strstr` returns a pointer to the beginning of the first occurrence of *string2* in *string1* . If *string2* does not appear in *string1* , `strstr` returns `NULL`. If *string2* points to a string with zero length, `strstr` returns *string1* .

Example Code

This example locates the string `haystack` in the string "`needle in a haystack`".

```
#include <string.h>

int main(void)
{
   char *string1 = "needle in a haystack";
   char *string2 = "haystack";
   char *result;

   result = strstr(string1, string2);

   /* Result = a pointer to "haystack"                                       */

   printf("%s\n", result);
   return 0;

   /****************************************************************************
      The output should be:

      haystack
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- strspn
- wcschr
- wcscspn
- wcspbrk
- wcsrchr
- wcsspn
- wcswcs

_strtime - Copy Time

_strtime - Copy Time

Syntax

```
#include <time.h>
char *_strtime(char *time);
```

Description

_strtime copies the current time into the buffer that *time* points to. The format is:

```
   hh:mm:ss
```

where

hh represents the hour in 24-hour notation,
mm represents the minutes past the hour,
ss represents the number of seconds.

For example, the string `18:23:44` represents 23 minutes and 44 seconds past 6 p.m.

The buffer must be at least 9 bytes.

Note: The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Returns

_strtime returns a pointer to the buffer. There is no error return.

Example Code

This example prints the current time:

```
#include <stdio.h>
#include <time.h>

int main(void)
{
   char buffer[9];

   printf("The current time is %s \n", _strtime(buffer));
   return 0;

   /****************************************************************************
      The output should be similar to:

      The current time is 16:47:22
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- localtime
- mktime
- time
- tzset

----

strtod - Convert Character String to Double

strtod - Convert Character String to Double

Syntax

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Description

`strtod` converts a character string to a double-precision value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type `double`. This function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the null character at the end of the string.

The `strtod` function expects *nptr* to point to a string with the following form:

```
>>                                                                    >
         white-space     +      digits
                                          .      digits
                                     .  digits

>                                ><
         e             digits
         E     +
```

The first character that does not fit this form stops the scan. The radix character is defined in the program's locale by the category LC_NUMERIC.

Returns

`strtod` returns the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, it returns `-HUGE_VAL` or `+HUGE_VAL`; for an underflow, it returns `0`.

In both cases, `errno` is set to `ERANGE`, depending on the base of the value. If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a `NULL` pointer.

`strtod` does not fail if a character other than a digit follows an `E` or `e` read in as an exponent. For example, `100elf` will be converted to the floating-point value `100.0`.

Example Code

This example converts the strings to a `double` value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string,*stopstring;
   double x;

   string = "3.1415926This stopped it";
   x = strtod(string, &stopstring);
   printf("string = %s\n", string);
   printf("   strtod = %f\n", x);
   printf("   Stopped scan at %s\n\n", stopstring);
   string = "100ergs";
   x = strtod(string, &stopstring);
   printf("string = \"%s\"\n", string);
   printf("   strtod = %f\n", x);
   printf("   Stopped scan at \"%s\"\n\n", stopstring);
   return 0;

   /****************************************************************************
```

```
      The output should be:

      string = 3.1415926This stopped it
         strtod = 3.141593
         Stopped scan at This stopped it

      string = "100ergs"
         strtod = 100.000000
         Stopped scan at "ergs"
   *****************************************************************************/
}
```

Related Information

- atof
- atoi
- atol
- _atold
- strtol
- strtold
- strtoul
- wcstod
- wcstol
- wcstoul

----

strtok - Tokenize String

strtok - Tokenize String

Syntax

```
#include <string.h>
char *strtok(char *string1, const char *string2);
```

Description

`strtok` reads *string1* as a series of zero or more tokens, and *string2* as the set of bytes serving as delimiters of the tokens in *string1* . The tokens in *string1* can be separated by one or more of the delimiters from *string2* . The tokens in *string1* can be located by a series of calls to `strtok`.

In the first call to `strtok` for a given *string1* , `strtok` searches for the first token in *string1* , skipping over leading delimiters. A pointer to the first token is returned.

To read the next token from *string1* , call `strtok` with a `NULL` *string1* argument. A `NULL` *string1* argument causes `strtok` to search for the next token in the previous token string. Each delimiter is replaced by a null character. The set of delimiters can vary from call to call, so *string2* can take any value.

Returns

The first time `strtok` is called, it returns a pointer to the first token in *string1* . In later calls with the same token string, `strtok` returns a pointer to the next token in the string. A `NULL` pointer is returned when there are no more tokens. All tokens are null-terminated.

Example Code

Using a loop, this example gathers tokens, separated by blanks or commas, from a string until no tokens are left. After processing the tokens (not shown), the example returns the pointers to the tokens `a`,`string`, `of`, and `tokens`. The next call to `strtok` returns `NULL`, and the loop ends.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *token,*string = "a string, of, ,tokens\0,after null terminator";

    /* the string pointed to by string is broken up into the tokens
       "a string", " of", " ", and "tokens" ; the null terminator (\0)
       is encountered and execution stops after the token "tokens"            */

   token = strtok(string, ",");
   do {
      printf("token: %s\n", token);
   }  while (token = strtok(NULL, ","));
   return 0;

   /****************************************************************************
      The output should be:

      token: a string
      token:  of
      token:
      token: tokens
   ****************************************************************************/
}
```

Related Information

- strcat
- strchr
- strcmp
- strcpy
- strcspn
- strspn
- wcstok

----

strtol - Convert Character String to Long Integer

strtol - Convert Character String to Long Integer

Syntax

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Description

`strtol` converts a character string to a long-integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of type `long int`. This function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the null character (`\0`) at the end of the string. The ending character can also be the first numeric character greater than or equal to the *base* .

When you use the `strtol` function, *nptr* should point to a string with the following form:

```
  >>                                                >< 
        white-space     +     0      digits
                              0x
                                0X
```

If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is `0`, the prefix determines the base (8, 16, or `10`): the prefix `0` means base 8 (octal); the prefix `0`x or `0`X means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Returns

`strtol` returns the value represented in the string, except when the representation causes an overflow. For an overflow, it returns `LONG_MAX` or `LONG_MIN`, according to the sign of the value and `errno` is set to `ERANGE`. If *base* is not a valid number, `strtol` sets `errno` to `EDOM`.

`errno` is set to `ERANGE` for the exceptional cases, depending on the base of the value. If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr* , provided that *endptr* is not a `NULL` pointer.

Example Code

This example converts the strings to a `long` value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string,*stopstring;
   long l;
   int bs;

   string = "10110134932";
   printf("string = %s\n", string);
   for (bs = 2; bs <= 8; bs *= 2) {
      l = strtol(string, &stopstring, bs);
      printf("   strtol = %ld (base %d)\n", l, bs);
      printf("   Stopped scan at %s\n\n", stopstring);
   }
   return 0;

   /****************************************************************************
      The output should be:

      string = 10110134932
         strtol = 45 (base 2)
         Stopped scan at 34932

      strtol = 4423 (base 4)
      Stopped scan at 4932

      strtol = 2134108 (base 8)
      Stopped scan at 932
   ****************************************************************************/
}
```

Related Information

- atof
- atoi
- atol
- _atold
- _ltoa
- strtod
- strtold
- strtoul
- wcstod
- wcstol
- wcstoul

----

strtold - Convert String to Long Double

strtold - Convert String to Long Double

Syntax

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr);
```

Description

strtold converts a character string pointed to by *nptr* to a long double value. When it reads a character it does not recognize as part of a number, strtold stops conversion and *endptr* points to the remainder of *nptr*. This character may be the ending null character.

The string pointed to by *nptr* must have the following format:

```
>>                                                                >
     whitespace                    digits
                          +                          .       digits
                                   .  digits

>                                 ><
      e              digits
      E       +
```

The *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one digit must follow the decimal point. An exponent expressed as a decimal integer can follow the digits. The exponent can be signed.

The value of *nptr* can also be one of the strings `infinity`, `inf`, or `nan`. These strings are case-insensitive, and can be preceded by a unary minus (`-`). They are converted to infinity and NaN values. See Infinity and NaN Support for more information about using infinity and NaN values.

If the string pointed to by *nptr* does not have the expected form, no conversion is performed and *endptr* points to the value of *nptr*.

The strtold function ignores any white-space characters, as defined by the `isspace` function.

Returns

If successful, strtold returns the value of the long double number. If it fails, strtold returns `0`. For an underflow or overflow, it returns the following:

Condition	**Return Value**
Underflow	`0` with `errno` set to ERANGE
Positive overflow	`+_LHUGE_VAL`
Negative overflow	`-_LHUGE_VAL`.

Example Code

This example uses strtold to convert two strings, `" -001234.5678e10end of string"` and `"NaNQthis cannot be converted"` to their corresponding long double values. It also prints out the part of the string that cannot be converted.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *nptr;
   char *endptr;
   nptr = "  -001234.5678e10end of string";
   printf("strtold = %.10Le\n", strtold(nptr, &endptr));
   printf("end pointer at = %s\n\n", endptr);
   nptr = "NaNthis cannot be converted";
   printf("strtold = %.10Le\n", strtold(nptr, &endptr));
   printf("end pointer at = %s\n\n", endptr);
   return 0;

   /****************************************************************************
      The output should be:

      strtold = -1.2345678000e+13
      end pointer at = end of string

      strtold = nan
      end pointer at = this cannot be converted
   ****************************************************************************/
}
```

Related Information

- atof
- atoi
- atol
- _atold
- strtod
- strtol
- strtoul
- wcstod
- wcstol
- wcstoul
- Infinity and NaN Support

----

strtoul - Convert String Segment to Unsigned Integer

strtoul - Convert String Segment to Unsigned Integer

Syntax

```
#include <stdlib.h>
unsigned long int strtoul(const char *string1, char **string2, int base);
```

Description

strtoul converts a character string to an unsigned long integer value. The input *string1* is a sequence of characters that can be interpreted as a numerical value of the type unsigned long int. strtoul stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. strtoul sets *string2* to point to the resulting output string if a conversion is performed, and provided that *string2* is not a NULL pointer.

When you use strtoul, *string1* should point to a string with the following form:

```
>>                                              ><
      white-space     +      0      digits
                             0x
                               0X
```

If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is `0`, the prefix determines the base (8, 16, or 10): the prefix `0` means base 8 (octal); the prefix `0x` or `0X` means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Returns

`strtoul` returns the value represented in the string, or `0` if no conversion could be performed. For an overflow, `strtoul` returns `ULONG_MAX` and sets `errno` to `ERANGE`. If *base* is not a valid number, `strtoul` sets errno to `EDOM`.

Example Code

This example converts the string to an unsigned long value. It prints out the converted value and the substring that stopped the conversion.

```
#include <stdio.h>
#include <stdlib.h>

#define  BASE          2

int main(void)
{
   char *string,*stopstring;
   unsigned long ul;

   string = "1000e13 e";
   printf("string = %s\n", string);
   ul = strtoul(string, &stopstring, BASE);
   printf("   strtoul = %ld (base %d)\n", ul, BASE);
   printf("   Stopped scan at %s\n\n", stopstring);
   return 0;

   /****************************************************************************
      The output should be:

      string = 1000e13 e
         strtoul = 8 (base 2)
         Stopped scan at e13 e
   ****************************************************************************/
}
```

Related Information

- atof
- atoi
- atol
- _atold
- strtod
- strtol
- strtold
- wcstod
- wcstol
- wcstoul

----

strupr - Convert Lowercase to Uppercase

strupr - Convert Lowercase to Uppercase

Syntax

```
#include <string.h>
char *strupr(char *string);
```

Description

strupr converts any lowercase letters in *string* to uppercase. Other characters are not affected.

Returns

strupr returns a pointer to the converted *string*. There is no error return.

Example Code

This example makes a copy in all uppercase of the string "DosWrite", and then prints the copy.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *string = "DosWrite";
   char *copy;

   copy = strupr(strdup(string));
   printf("This is a copy of the string\n");
   printf("with all letters capitalized: %s\n", copy);
   return 0;

   /****************************************************************************
      The output should be:

      This is a copy of the string
      with all letters capitalized: DOSWRITE
   ****************************************************************************/
}
```

Related Information

- strlwr
- _toascii - _tolower - _toupper
- tolower - toupper
- towlower - towupper

----

strxfrm - Transform String

strxfrm - Transform String

Syntax

```
#include <string.h>
size_t strxfrm(char *str1, const char *str2, size_t n);
```

Description

strxfrm transforms the string pointed to by *str2* and places the resulting string into the array pointed to by *str1*. The transformation is determined by the program's locale. The transformed string is not necessarily readable, but can be used with the strcmp or strncmp functions. qsort and bsearch can also be used on the results.

The transformation is such that, if strcmp or strncmp were applied to the two transformed strings, the results would be the same as applying strcoll to the two corresponding untransformed strings.

No more than *n* bytes are placed into the area pointed to by *str1*, including the terminating null byte. If *n* is 0, *str1* can be a null pointer.

Returns

strxfrm returns the length of the transformed string (excluding the null byte). When *n* is `0` and *str1* is a null pointer, the length returned is one less than the number of bytes required to contain the transformed string. If an error occurs, strxfrm function returns (size_t)-1 and sets errno to indicate the error.

Note:

- The string returned by strxfrm may be longer than the input string; it does not contain printable characters.
- strxfrm calls malloc when the LC_COLLATE category specifies *backward* on the *order_start* keyword, the *substitute* keyword is specified, or the locale has one-to-many mapping. If malloc fails, strxfrm also fails.

Example Code

This example uses strxfrm to transform two different strings that have the same collating weight. It then calls strcmp to compare the new strings.

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>

int main(void)
{
   char *string1 = "stride ng1";
   char *string2 = "stri*ng1";
   char *newstring1, *newstring2;
   int  length1, length2;

   if (NULL == setlocale(LC_ALL, "Fr_FR")) {
      printf("setlocale failed.\n");
      exit(EXIT_FAILURE);
   }
   length1=strxfrm(NULL, string1, 0);
   length2=strxfrm(NULL, string2, 0);
   if (NULL == (newstring1 = calloc(length1 + 1, 1)) ||
       NULL == (newstring2 = calloc(length2 + 1, 1))) {
      printf("insufficient memory\n");
      exit(EXIT_FAILURE);
   }
   if ((strxfrm(newstring1, string1, length1 + 1) != length1) ||
       (strxfrm(newstring2, string2, length2 + 1) != length2)) {
      printf("error in string processing\n");
      exit(EXIT_FAILURE);
   }
   if (0 != strcmp(newstring1, newstring2))
      printf("wrong results\n");
   else
      printf("correct results\n");
   return 0;

   /****************************************************************************
      The output should be similar to :

      correct results
   ****************************************************************************/
}
```

Related Information

- localeconv
- setlocale
- strcmp
- strcoll
- strncmp
- wcsxfrm

----

swab - Swap Adjacent Bytes

swab - Swap Adjacent Bytes

Syntax

```
#include <stdlib.h>
void swab(char *source, char *destination, int n);
```

Description

swab copies *n* bytes from *source* , swaps each pair of adjacent bytes, and stores the result at *destination* . The integer *n* should be an even number to allow for swapping. If *n* is an odd number, a null character (`\0`) is added after the last byte.

swab is typically used to prepare binary data for transfer to a machine that uses a different byte order.

Note: In earlier releases of C Set ++, swab began with an underscore (`_swab`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_swab` to swab for you.

Returns

There is no return value.

Example Code

This example copies *n* bytes from one location to another, swapping each pair of adjacent bytes:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char from[20] = "hTsii  s atsirgn..x ";
   char to[20];

   swab(from, to, 19);  /* swap bytes */
   printf("%s\n", to);

   return 0;

   /****************************************************************************
      The output should be:

      This is a string..
   ****************************************************************************/
}
```

Related Information

- fgetc
- fputc

----

system - Invoke the Command Processor

system - Invoke the Command Processor

Syntax

```
#include <stdlib.h>
int system(char *string);
```

Description

system passes the command *string* to a command processor to be run. The command processor specified in the COMSPEC environment variable is first searched for. If it does not exist or is not a valid executable file, the default command processor, `CMD.EXE`, is searched for in the current directory and in all the directories specified in the PATH environment variable.

If the specified command is the name of an executable file created from a C program, full initialization and termination are performed, including automatic flushing of buffers and closing of files. To pass information across a system function, use a method of interprocess communication like pipes or shared memory.

You can also use system to redirect standard streams using the redirection operators (the angle brackets), for example:

```
   rc = system("cprogram < in.file");
```

The defaults for the standard streams will be whatever the standard streams are at the point of the system call; for example, if the root program redirects stdout to `file.txt`, a `printf` call in a C module invoked by a system call will append to `file.txt`.

Returns

If the argument is a null pointer, system returns nonzero if a command processor exists, and `0` if it does not. system returns the the return value from the command processor if it is successfully called. If system cannot call the command processor successfully, the return value is -1 and `errno` is set to one of the following values: compact break=fit.

Value	**Meaning**
ENOCMD	No valid command processor found.
ENOMEM	Insufficient memory to complete the function.
EOS2ERR	A system error occurred. Check _doserrno for the specific OS/2 error code.

Example Code

This example shows how to use system to execute the OS/2 command `dir c:\` :

```
#include <stdlib.h>

int main(void)
{
   int rc;

   rc = system("dir c:\\");
   return rc;

   /* The output should be a listing of the root directory on the c: drive    */
}
```

Related Information

- exit
- _exit

----

tan - Calculate Tangent

tan - Calculate Tangent

Syntax

```
#include <math.h>
double tan(double x);
```

Description

`tan` calculates the tangent of *x*, where *x* is expressed in radians. If *x* is too large, a partial loss of significance in the result can occur.

Returns

`tan` returns the value of the tangent of *x*.

Example Code

This example computes *x* as the tangent of pi/4.

```
#include <math.h>

int main(void)
{
   double pi,x;

   pi = 3.1415926;
   x = tan(pi/4.0);
   printf("tan( %lf ) is %lf\n", pi/4, x);
   return 0;

   /****************************************************************************
      The output should be:

      tan( 0.785398 ) is 1.000000
   ****************************************************************************/
}
```

Related Information

- atan - atan2
- cos
- sin
- tanh

----

tanh - Calculate Hyperbolic Tangent

tanh - Calculate Hyperbolic Tangent

Syntax

```
#include <math.h>
double tanh(double x);
```

Description

`tanh` calculates the hyperbolic tangent of *x*, where *x* is expressed in radians.

Returns

`tanh` returns the value of the hyperbolic tangent of *x*. The result of `tanh` cannot have a range error.

Example Code

This example computes *x* as the hyperbolic tangent of pi/4.

```
#include <math.h>

int main(void)
{
   double pi,x;

   pi = 3.1415926;
   x = tanh(pi/4);
   printf("tanh( %lf ) = %lf\n", pi/4, x);
   return 0;

   /****************************************************************************
      The output should be:

      tanh( 0.785398 ) = 0.655794

   ****************************************************************************/
}
```

Related Information

- atan - atan2
- cosh
- sinh
- tan

_tell - Get Pointer Position

_tell - Get Pointer Position

Syntax

```
#include <io.h>
long _tell(int handle);
```

Description

_tell gets the current position of the file pointer associated with *handle* . The position is the number of bytes from the beginning of the file.

Returns

_tell returns the current position. A return value of `-1L` indicates an error, and `errno` is set to one of the following values: compact break=fit.

Value	**Meaning**
EBADF	The file handle is not valid.

EOS2ERR	The call to the operating system was not successful.

On devices incapable of seeking (such as screens and printers), the return value is `-1L`.

Example Code

This example opens the file `tell.dat`. It then calls _tell to get the current position of the file pointer and report whether the operation is successful. The program then closes `tell.dat`.

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#define FILENAME  "tell.dat"

int main(void)
{
   long filePtrPos;
   int fh;

   printf("Creating file.\n");
   system("echo Sample Program > " FILENAME);
   if (-1 == (fh = open(FILENAME, O_RDWR|O_APPEND))) {
      perror("Unable to open " FILENAME);
      remove(FILENAME);
      return EXIT_FAILURE;
   }
   /* Get the current file pointer position.                                 */
   if (-1 == (filePtrPos = _tell(fh))) {
      perror("Unable to tell");
      close(fh);
      remove(FILENAME);
      return EXIT_FAILURE;
   }
   printf("File pointer is at position %d.\n", filePtrPos);
   close(fh);
   remove(FILENAME);
   return 0;

   /****************************************************************************
      The output should be:

      Creating file.
      File pointer is at position 0.
   ****************************************************************************/
}
```

Related Information

- fseek
- ftell
- lseek

----

tempnam - Produce Temporary File Name

tempnam - Produce Temporary File Name

Syntax

```
#include <stdio.h>
char *tempnam(char *dir, char *prefix);
```

Description

tempnam creates a temporary file name in another directory. The *prefix* is the prefix to the file name. tempnam tests for the existence of the file with the given name in the following directories, listed in order of precedence:

- If the TMP environment variable is set and the directory specified by TMP exists, the directory is specified by TMP.

- If the TMP environment variable is not set or the directory specified by TMP does not exist, the directory is specified by the *dir* argument to tempnam.

- If the *dir* argument is NULL, or *dir* is the name of nonexistent directory, the directory is pointed to by `P_tmpdir` (defined in `<stdio.h>`).

- If `P_tmpdir` does not exist, the directory is the current working directory.

Note:

- Because tempnam uses malloc to reserve space for the created file name, you must free this space when you no longer need it.

- In earlier releases of C Set ++, tempnam began with an underscore (`_tempnam`). Because it is defined by the X/Open standard, the underscore has been removed. For compatibility, *The Developer's Toolkit* will map `_tempnam` to tempnam for you.

Returns

tempnam returns a pointer to the temporary name, if successful. If the name cannot be created or it already exists, tempnam returns the value `NULL`.

Example Code

This example creates a temporary file name using the directory `a:\tmp`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *name1;

   if ((name1 = tempnam("d:\\tmp", "stq")) != NULL)
      printf("%s is safe to use as a temporary file.\n", name1);
   else {
      printf("Cannot create unique filename\n");
      return EXIT_FAILURE;
   }
   return 0;

   /***************************************************************************
      The output should be similar to:

      D:\tmp\stqU3CP2.C2T is safe to use as a temporary file.
   ***************************************************************************/
}
```

Related Information

- malloc
- _rmtmp
- tmpfile
- tmpnam

_threadstore - Access Thread-Specific Storage

_threadstore - Access Thread-Specific Storage

Syntax

```
#include <stdlib.h>
void *_threadstore(void);
```

Description

_threadstore provides access to a private thread pointer that is initialized to `NULL`. You can assign any thread-specific data structure to this pointer.

You can only use _threadstore in multithread programs.

Returns

_threadstore returns the address of the pointer to the defined thread storage space.

Example Code

This example uses _threadstore to point to storage that is local to the thread. It prints the address pointed to by _threadstore.

```
#include <stdlib.h>
#include <stdio.h>

#define  privateStore   (*_threadstore())

void thread(void *dummy)
{
   privateStore = malloc(100);
   printf("The starting address of the storage space is %p\n", privateStore);

   /* user must free storage before exiting thread */
   free (privateStore);
   _endthread();
}

int main(void)
{
   int i;

   for (i = 0; i < 10; i++)
      _beginthread(thread, NULL, (unsigned) 32096, NULL);
   DosSleep(5000L);
   return 0;
}
```

Related Information

- _beginthread
- _endthread

----

time - Determine Current Time

time - Determine Current Time

Syntax

```
#include <time.h>
```

```
time_t time(time_t *timeptr);
```

Description

`time` determines the current calendar time, which is not necessarily the local time `localtime`.

Note: The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Returns

`time` returns the current calendar time. The return value is also stored in the location given by *timeptr*. If *timeptr* is `NULL`, the return value is not stored. If the calendar time is not available, the value `(time_t)(-1)` is returned.

Example Code

This example gets the time and assigns it to *ltime*. `ctime` then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;

   time(&ltime);
   printf("The time is %s\n", ctime(&ltime));
   return 0;

   /****************************************************************************
      The output should be similar to:

      The time is Thu Jan 12 11:38:37 1995
   ****************************************************************************/
}
```

Related Information

- asctime
- ctime
- gmtime
- localtime
- mktime
- _strtime

----

tmpfile - Create Temporary File

tmpfile - Create Temporary File

Syntax

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

`tmpfile` creates a temporary binary file. The file is automatically removed when it is closed or when the program is terminated.

`tmpfile` opens the temporary file in `wb+` mode.

Returns

`tmpfile` returns a stream pointer, if successful. If it cannot open the file, it returns a `NULL` pointer. On normal termination (`exit`), these temporary files are removed.

Example Code

This example creates a temporary file, and if successful, writes `tmpstring` to it. At program termination, the file is removed.

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
char tmpstring[] = "This is the string to be temporarily written";

int main(void)
{
   if (NULL == (stream = tmpfile())) {
      perror("Cannot make a temporary file");
      return EXIT_FAILURE;
   }
   else
      fprintf(stream, "%s", tmpstring);
   return 0;
}
```

Related Information

- fopen
- tmpnam
- tempnam
- _rmtmp

----

tmpnam - Produce Temporary File Name

tmpnam - Produce Temporary File Name

Syntax

```
#include <stdio.h>
char *tmpnam(char *string);
```

Description

`tmpnam` produces a valid file name that is not the same as the name of any existing file. It stores this name in *string*. If *string* is `NULL`, `tmpnam` leaves the result in an internal static buffer. Any subsequent calls destroy this value. If *string* is not `NULL`, it must point to an array of at least `L_tmpnam` bytes. The value of `L_tmpnam` is defined in `<stdio.h>`.

`tmpnam` produces a different name each time it is called within a module up to at least `TMP_MAX` (a value of at least 25) names. Note that files created using names returned by `tmpnam` are not automatically discarded at the end of the program. Files can be removed by the `remove` function.

Returns

`tmpnam` returns a pointer to the name. If it cannot create a unique name; it returns `NULL`.

Example Code

This example calls `tmpnam` to produce a valid file name.

```
#include <stdio.h>

int main(void)
{
   char *name1;

   if ((name1 = tmpnam(NULL)) != NULL)
      printf("%s can be used as a file name.\n", name1);
   else
      printf("Cannot create a unique file name\n");

   return 0;

   /****************************************************************************
      The output should be similar to:

      d:\tmp\acc00000.CTN can be used as a file name.

   ****************************************************************************/
}
```

Related Information

- fopen
- remove
- tempnam

_toascii - _tolower - _toupper - Convert Character

_toascii - _tolower - _toupper - Convert Character

Syntax

```
#include <ctype.h>
int _toascii(int c);
int _tolower(int c);
int _toupper(int c);
```

Description

_toascii converts *c* to a character in the ASCII character set, by setting all but the low-order 7 bits to `0`. If *c* already represents an ASCII character, _toascii does not change it.

_tolower converts *c* to the corresponding lowercase letter, if possible.

_toupper converts *c* to the corresponding uppercase letter, if possible.

Important Use _tolower and _toupper only when you know that *c* is uppercase A to Z or lowercase a to z, respectively. Otherwise the results are undefined. These functions are not affected by the current locale.

These are all macros, and do not correctly handle arguments with side effects.

For portability, use the tolower and toupper functions defined by the ANSI/ISO standard, instead of the _tolower and _toupper macros.

Returns

_toascii, _tolower, and _toupper return the possibly converted character *c*. If the character passed to _toascii is an ASCII character, _toascii returns the character unchanged. There is no error return.

Example Code

This example prints four sets of characters. The first set is the ASCII characters having graphic images, which range from `0x21` through `0x7e`. The second set takes integers `0x7f21` through `0x7f7e` and applies the _toascii macro to them, yielding the same set of printable characters. The third set is the characters with all lowercase letters converted to uppercase. The fourth set is the characters with all uppercase letters converted to lowercase.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   printf("Characters 0x01 to 0x03, and integers 0x7f01 to 0x7f03 mapped to\n");
   printf("ASCII by _toascii() both yield the same graphic characters.\n\n");
   for (ch = 0x01; ch <= 0x03; ch++) {
      printf("char 0x%.4X: %c    ", ch, ch);
      printf("char _toascii(0x%.4X): %c\n", ch+0x7f00, ch+0x7f00);
   }
   printf("\nCharacters A, B and C converted to lowercase and\n");
   printf("Characters a, b and c converted to uppercase.\n\n");
   for (ch = 0x41; ch <= 0x43; ch++) {
      printf("_tolower(%c) = %c   ", ch, _tolower(ch));
      printf("_toupper(%c) = %c\n", ch+0x20, _toupper(ch+0x20));
   }
   return 0;

   /***************************************************************************
      The output should be:

      Characters 0x01 to 0x03, and integers 0x7f01 to 0x7f03 mapped to
      ASCII by _toascii() both yield the same graphic characters.

      char 0x0001:      char toascii(0x7F01):
      char 0x0002:      char toascii(0x7F02):
      char 0x0003:      char toascii(0x7F03):

      Characters A, B and C converted to lowercase and
      Characters a, b and c converted to uppercase.

      _tolower(A) = a   _toupper(a) = A
      _tolower(B) = b   _toupper(b) = B
      _tolower(C) = c   _toupper(c) = C
   ***************************************************************************/
}
```

Related Information

- isalnum to isxdigit
- isascii
- _iscsym - _iscsymf
- tolower - toupper
- towlower - towupper

----

tolower - toupper - Convert Character Case

tolower - toupper - Convert Character Case

Syntax

```
#include <ctype.h>
int tolower(int C);
int toupper(int c);
```

Description

`tolower` converts the uppercase letter *C* to the corresponding lowercase letter.

`toupper` converts the lowercase letter *c* to the corresponding uppercase letter.

The character mapping is determined by the LC_CTYPE category of the current locale.

Note: `toupper` and `tolower` can only be used for single-byte characters. `towupper` and `towlower` should be used for case conversion of wide characters that are equivalent to both single-byte and double-byte characters.

Returns

Both functions return the converted character. If the character *c* does not have a corresponding lowercase or uppercase character, the functions return *c* unchanged.

Example Code

This example uses `toupper` and `tolower` to modify characters between code `0` and code `7f`.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   /* print hex values of characters */
   for (ch = 0; ch <= 0x7f; ch++) {
      printf("toupper=%#04x\n", toupper(ch));
      printf("tolower=%#04x\n", tolower(ch));
      putchar('\n');
   }
   return 0;
}
```

Related Information

- isalnum to isxdigit
- isascii
- _toascii - _tolower - _toupper
- towlower - towupper

----

towlower - towupper - Convert Wide Character Case

towlower - towupper - Convert Wide Character Case

Syntax

```
#include <wctype.h>
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

Description

towlower converts the uppercase letter *wc* to the corresponding lowercase letter.

towupper converts the lowercase letter *wc* to the corresponding uppercase letter.

The character mapping is determined by the LC_CTYPE category of current locale.

Returns

Both functions return the converted character. If the wide character *wc* does not have a corresponding lowercase or uppercase character, the functions return *wc* unchanged.

Example Code

This example uses towlower and towupper to convert characters between `0` and `0x7f`.

```
#include <wchar.h>
#include <stdio.h>

int main(void)
{
   wint_t w_ch;

   for (w_ch = 0; w_ch <= 0x7f; w_ch++) {
      printf ("towupper : %#04x %#04x, ", w_ch, towupper(w_ch));
      printf ("towlower : %#04x %#04x\n", w_ch, towlower(w_ch));
   }
   return 0;

   /*****************************************************************************
      The output should be similar to :
      .
      :
      towupper : 0x41 0x41, towlower : 0x41 0x61
      towupper : 0x42 0x42, towlower : 0x42 0x62
      towupper : 0x43 0x43, towlower : 0x43 0x63
      towupper : 0x44 0x44, towlower : 0x44 0x64
      towupper : 0x45 0x45, towlower : 0x45 0x65
      .
      :
      towupper : 0x61 0x41, towlower : 0x61 0x61
      towupper : 0x62 0x42, towlower : 0x62 0x62
      towupper : 0x63 0x43, towlower : 0x63 0x63
      towupper : 0x64 0x44, towlower : 0x64 0x64
      towupper : 0x65 0x45, towlower : 0x65 0x65
      :
   *****************************************************************************/
}
```

Related Information

- tolower - toupper
- _toascii - _tolower - _toupper

----

tzset - Set Time Zone Information

tzset - Set Time Zone Information

Syntax

```
#include <time.h>
```

```
void tzset(void);
```

Description

tzset uses the environment variable TZ to change the time zone and daylight saving time (DST) zone values. These values are used by the `gmtime` and `localtime` functions to make corrections from Coordinated Universal Time (formerly GMT) to local time.

To use tzset, set the TZ variable to the appropriate values. (For the possible values for TZ, see the chapter on run-time environment variables in the *VisualAge C++ Programming Guide* .) Then call tzset to incorporate the changes in the time zone information into your current locale.

To set TZ from within a program, use putenv before calling tzset.

Note: The time and date functions begin at 00:00:00 Coordinated Universal Time, January 1, 1970.

Returns

There is no return value.

Example Code

This example uses putenv and tzset to set the time zone to Central Time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t currentTime;
   struct tm *ts;

   /* Get the current time                                                 */

   (void)time(&currentTime);
   printf("The GMT time is %s", asctime(gmtime(&currentTime)));
   ts = localtime(&currentTime);
   if (ts->tm_isdst > 0)  /* check if Daylight Saving Time is in effect       */
      {
      printf("The local time is %s", asctime(ts));
      printf("Daylight Saving Time is in effect.\n");
   }
   else {
      printf("The local time is %s", asctime(ts));
      printf("Daylight Saving Time is not in effect.\n");
   }
   printf("**** Changing to Central Time ****\n");
   putenv("TZ=CST6CDT");
   tzset();
   ts = localtime(&currentTime);
   if (ts->tm_isdst > 0)  /* check if Daylight Saving Time is in effect       */
      {
      printf("The local time is %s", asctime(ts));
      printf("Daylight Saving Time is in effect.\n");
   }
   else {
      printf("The local time is %s", asctime(ts));
      printf("Daylight Saving Time is not in effect.\n");
   }

   return 0;

   /****************************************************************************
      The output should be similar to:

      The GMT time is Fri Jan 13 21:49:26 1995
      The local time is Fri Jan 13 16:49:26 1995
      Daylight Saving Time is not in effect.
      **** Changing to Central Time ****
      The local time is Fri Jan 13 15:49:26 1995
      Daylight Saving Time is not in effect.
   ****************************************************************************/
}
```

Related Information

- asctime
- _ftime
- gmtime
- localtime
- mktime
- putenv
- strftime
- time

_uaddmem - Add Memory to a Heap

_uaddmem - Add Memory to a Heap

Syntax

```
#include <umalloc.h>
Heap_t _uaddmem(Heap_t heap, void *block, size_t size, int clean);
```

Description

_uaddmem adds a *block* of memory of *size* bytes into the specified user *heap* (created with _ucreate). Before calling _uaddmem, you must first get the *block* from the operating system, typically by using an OS/2 function like DosAllocMem or by allocating it statically. (See the *Control Program Guide and Reference* for details on OS/2 functions for memory management.)

If the memory *block* has been initialized to `0`, specify `_BLOCK_CLEAN` for the *clean* parameter. If not, specify `!_BLOCK_CLEAN`. (This information makes calloc and _ucalloc more efficient).

Note: Memory returned by DosAllocMem is initialized to `0`.

For fixed-size heaps, you must return all the blocks you added with _uaddmem to the system. (For expandable heaps, these blocks are returned by your *release_fn* when you call _udestroy.)

For more information about creating and using heaps, see "Managing Memory" in the *VisualAge C++ Programming Guide* .

Note: For every block of memory you add, a small number of bytes from it are used to store internal information. To reduce the total amount of overhead, it is better to add a few large blocks of memory than many small blocks.

Returns

_uaddmem returns a pointer to the heap the memory was added to. If the heap specified is not valid, _uaddmem returns NULL.

Example Code

The following example creates a heap `myheap`, and then uses _uaddmem to add memory to it.

```
#define  INCL_DOSMEMMGR             /* Memory Manager values */
#include <os2.h>
#include <bsememf.h>                /* Get flags for memory management  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
```

```
int main(void)
{
   void    *initial_block, *extra_chunk;
   APIRET  rc;
   Heap_t  myheap;
   char    *p1, *p2;

   /* Call DosAllocMem to get the initial block of memory */
   if (0 != (rc = DosAllocMem(&initial_block, 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem for initial block failed: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   /* Create a fixed size heap starting with the block declared earlier */
   if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                  _HEAP_REGULAR, NULL, NULL))) {
      puts("_ucreate failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != _uopen(myheap)) {
      puts("_uopen failed.");
      exit(EXIT_FAILURE);
   }
   p1 = _umalloc(myheap, 100);
   /* Call DosAllocMem to get another block of memory */
   if (0 != (rc = DosAllocMem(&extra_chunk, 10 * 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem for extra chunk failed: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   /* Add the second chunk of memory to user heap */
   if (myheap != _uaddmem(myheap, extra_chunk, 10 * 65536, _BLOCK_CLEAN)) {
      puts("_uaddmem failed.");
      exit(EXIT_FAILURE);
   }
   p2 = _umalloc(myheap, 100000);
   free(p1);
   free(p2);
   if (0 != _uclose(myheap)) {
      puts("_uclose failed");
      exit(EXIT_FAILURE);
   }
   if (0 != DosFreeMem(initial_block) || 0 != DosFreeMem(extra_chunk)) {
      puts("DosFreeMem error.");
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- _ucreate
- _udestroy
- _uheapmin
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_ucalloc - Reserve and Initialize Memory from User Heap

_ucalloc - Reserve and Initialize Memory from User Heap

Syntax

```
#include <umalloc.h>
void *_ucalloc(Heap_t heap, size_t num, size_t size);
```

Description

_ucalloc allocates memory for an array of *num* elements, each of length *size* bytes, from the *heap* you specify. It then initializes all bits of each element to `0`.

_ucalloc works just like calloc except that you specify the heap to use; calloc always allocates from the default heap. If the *heap* does not have enough memory for the request, _ucalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To reallocate or free memory allocated with _ucalloc, use the non-heap-specific realloc and free. These functions always check what heap the memory was allocated from.

Returns

_ucalloc returns a pointer to the reserved space. If *size* or *num* was specified as zero, or if your *getmore_fn* cannot provide enough memory, _ucalloc returns NULL. Passing _ucalloc a heap that is not valid results in undefined behavior.

Example Code

This example creates a heap `myheap` and then uses _ucalloc to allocate memory from it.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = _ucalloc(myheap, 100, 1))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 10);
   free(ptr);
   return 0;
}
```

Related Information

- calloc
- free
- realloc
- _ucreate
- _umalloc
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_uclose - Close Heap from Use

_uclose - Close Heap from Use

Syntax

```
#include <umalloc.h>
int _uclose(Heap_t heap);
```

Description

_uclose closes a *heap* when a process will not use it again. After you close a heap, any attempt in the current process to allocate or return memory to it will have undefined results. _uclose affects only the current process; if the heap is shared, other processes may still be able to access it.

Once you have closed the heap, use _udestroy to destroy it and return all its memory to the operating system.

Note: If the heap is shared, you must close it in all processes that share it before you destroy it, or undefined results will occur.

You cannot close *The Developer's Toolkit* run-time heap (_RUNTIME_HEAP).

For more information about creating and using heaps, see "Managing Memory" in the *VisualAge C++ Programming Guide* .

Returns

If successful, _uclose returns 0. A nonzero return value indicates failure. Passing _uclose a heap that is not valid results in undefined behavior.

Example Code

The following example creates and opens a heap, and then performs operations on it. It then calls _uclose to close the heap before destroying it.

```
#define  INCL_DOSMEMMGR             /* Memory Manager values */
#include <os2.h>
#include <bsememf.h>                /* Get flags for memory management  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   void    *initial_block;
   APIRET  rc;
   Heap_t  myheap;
   char    *p;

   /* Call DosAllocMem to get the initial block of memory */
   if (0 != (rc = DosAllocMem(&initial_block, 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   /* Create a fixed size heap starting with the block declared earlier */
   if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                  _HEAP_REGULAR, NULL, NULL))) {
      puts("_ucreate failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != _uopen(myheap)) {
      puts("_uopen failed.");
      exit(EXIT_FAILURE);
   }
   p = _umalloc(myheap, 100);
   memset(p, 'x', 10);
   free(p);

   if (0 != _uclose(myheap)) {
      puts("_uclose failed");
      exit(EXIT_FAILURE);
   }
   if (0 != (rc = DosFreeMem(initial_block))) {
      printf("DosFreeMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- _uopen
- _udestroy
- _ucreate
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_ucreate - Create a Memory Heap

_ucreate - Create a Memory Heap

Syntax

```
#include <umalloc.h>
Heap_t _ucreate(void *block, size_t initsz, int clean, int memtype,
                void *(*getmore_fn)(Heap_t, size_t *, int *)
                void  (*release_fn)(Heap_t, void *, size_t);
```

Description

_ucreate creates your own memory heap that you can allocate and free memory from, just like *The Developer's Toolkit* run-time heap.

Before you call _ucreate, you must first get the initial *block* of memory for the heap. You can get this block by calling an OS/2 function (such as DosAllocMem or or DosAllocSharedMem) or by statically allocating it. (See the *CP Programming Guide and Reference* for more information on the OS/2 functions.)

Note: You must also return this initial block of memory to the system after you destroy the heap.

When you call _ucreate, you pass it the following parameters:

block
The pointer to the initial block you obtained.

initsz
The size of the initial block, which must be at least _HEAP_MIN_SIZE bytes (defined in `<malloc.h>`). If you are creating a fixed-size heap, the size must be large enough to satisfy all memory requests your program will make of it.

clean
The macro _BLOCK_CLEAN, if the memory in the block has been initialized to `0`, or !_BLOCK_CLEAN, if the memory has not been touched. This improves the efficiency of _ucalloc; if the memory is already initialized to `0`, _ucalloc does not need to initialize it.

Note: DosAllocMem initializes memory to `0` for you. You can also use memset to initialize the memory; however, memset also commits all the memory at once, an action that could slow overall performance.

memtype
A macro indicating the type of memory in your heap: _HEAP_REGULAR (regular) or _HEAP_SHARED (shared). Shared memory can be shared between different processes. For more information on different types of memory, see the *VisualAge C++ Programming Guide* and the *Control Program Guide and Reference* .

Note: Make sure that when you get the initial block, you request the same type of memory that you specify for *memtype* .

getmore_fn
A function you provide to get more memory from the system (typically using OS/2 functions or static allocation). To create a fixed-size heap, specify NULL for this parameter.

release_fn
A function you provide to return memory to the system (typically using DosFreeMem). To create a fixed-size heap, specify NULL for this parameter.

If you create a fixed-size heap, the initial block of memory must be large enough to satisfy all allocation requests made to it. Once the block is fully allocated, further allocation requests to the heap will fail. If you create an expandable heap, the *getmore_fn* and *release_fn* allow your heap to expand and shrink dynamically.

When you call _umalloc (or a similar function) for your heap, if not enough memory is available in the block, it calls the *getmore_fn* you provide. Your *getmore_fn* then gets more memory from the system and adds it to the heap, using any method you choose.

Your *getmore_fn* must have the following prototype:

```
   void *(*getmore_fn)(Heap_t uh, size_t *size, int *clean);
```

where: compact break=fit.

uh Is the heap to get memory for.

size Is the size of the allocation request passed by _umalloc. You probably want to return enough memory at a time to satisfy several allocations; otherwise, every subsequent allocation has to call *getmore_fn* . You should return multiples of 64K (the smallest size that DosAllocMem returns). Make sure you update the *size* parameter if you return more than the original request.

clean Within *getmore_fn* , you must set this variable either to _BLOCK_CLEAN, to indicate that you initialized the memory to `0`, or to !_BLOCK_CLEAN, to indicate that the memory is untouched.

Note: Make sure your *getmore_fn* allocates the right type of memory for the heap.

When you call _uheapmin to coalesce the heap or _udestroy to destroy it, these functions call the *release_fn* you provide to return the memory to the system. function.

Your *release_fn* must have the following prototype:

```
   void (*release_fn)(Heap_t uh, void *block, size_t size);
```

The heap *uh* the block is from, the *block* to be returned, and its *size* are passed to *release_fn* by _uheapmin or _udestroy.

For more information about creating and using heaps, see the "Managing Memory" in the *VisualAge C++ Programming Guide* .

Returns

If successful, _ucreate returns a pointer to the heap created. If errors occur, _ucreate returns NULL.

Example Code

The following example uses _ucreate to create an expandable heap. The functions for expanding and shrinking the heap are `get_fn` and `release_fn`. The program then opens the heap and performs operations on it, and then closes and destroys the heap.

```
#define  INCL_DOSMEMMGR             /* Memory Manager values */
#include <os2.h>
#include <bsememf.h>                /* Get flags for memory management  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
   void *p;

   /* Round up to the next chunk size */
   *length = ((*length) / 65536) * 65536 + 65536;
   *clean = _BLOCK_CLEAN;
   DosAllocMem(&p, *length, PAG_COMMIT | PAG_READ | PAG_WRITE);
   return (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
   DosFreeMem(p);
   return;
}

int main(void)
{
   void    *initial_block;
   APIRET  rc;
   Heap_t  myheap;
   char    *ptr;

   /* Call DosAllocMem to get the initial block of memory */
   if (0 != (rc = DosAllocMem(&initial_block, 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }



   /* Create an expandable heap starting with the block declared earlier */
   if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                  _HEAP_REGULAR, get_fn, release_fn))) {
      puts("_ucreate failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != _uopen(myheap)) {
      puts("_uopen failed.");
      exit(EXIT_FAILURE);
   }

   /* Force user heap to grow */
   ptr = _umalloc(myheap, 100000);

   _uclose(myheap);

   if (0 != _udestroy(myheap, _FORCE)) {
      puts("_udestroy failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != (rc = DosFreeMem(initial_block))) {
      printf("DosFreeMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- _uaddmem
- _ucalloc
- _uclose
- _udestroy
- _uheapmin
- _umalloc
- _uopen
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_udefault - Change the Default Heap

_udefault - Change the Default Heap

Syntax

```
#include <umalloc.h>
Heap_t _udefault(Heap_t heap);
```

Description

_udefault makes the *heap* you specify become the default heap. All calls to memory management functions that do not specify a heap (including malloc and calloc) then allocate memory from the *heap* .

This change affects only the thread where you called _udefault.

The initial default heap is *The Developer's Toolkit* run-time heap. To restore or explicitly set *The Developer's Toolkit* run-time heap as the default, call _udefault with the argument _RUNTIME_HEAP.

You can also use _udefault to find out which heap is being used as the default by specifying NULL for the *heap* parameter. The default heap remains the same.

For more information about creating and using heaps, see "Managing Memory" in the *VisualAge C++ Programming Guide* .

Returns

_udefault returns a pointer to the former default heap. You can save this pointer and use it later to restore the original heap. If the call is unsuccessful, _udefault returns NULL. Passing _udefault a heap that is not valid results in undefined behavior.

Example Code

This example creates a fixed-size heap myheap and uses _udefault to make it the default heap. The call to malloc then allocates memory from myheap. The second call to _udefault restores the original default heap.

```
#define  INCL_DOSMEMMGR             /* Memory Manager values */
#include <os2.h>
#include <bsememf.h>                /* Get flags for memory management  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   void    *initial_block;
   APIRET  rc;
   Heap_t  myheap, old_heap;
   char    *p;

   /* Call DosAllocMem to get the initial block of memory */
   if (0 != (rc = DosAllocMem(&initial_block, 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   /* Create a fixed size heap starting with the block declared earlier */
   if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                  _HEAP_REGULAR, NULL, NULL))) {
      puts("_ucreate failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != _uopen(myheap)) {
      puts("_uopen failed.");
      exit(EXIT_FAILURE);
   }

   /* myheap is used as default heap */
   old_heap = _udefault(myheap);

   /* malloc will allocate memory from myheap */
   p = malloc(100);
   memset(p, 'x', 10);

   /* Restore original default heap */
   _udefault(old_heap);

   free(p);
   if (0 != _uclose(myheap)) {
      puts("_uclose failed");
      exit(EXIT_FAILURE);
   }
   if (0 != (rc = DosFreeMem(initial_block))) {
      printf("DosFreeMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- calloc
- malloc
- _mheap
- _ucreate
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_udestroy - Destroy a Heap

_udestroy - Destroy a Heap

Syntax

```
#include <umalloc.h>
int _udestroy(Heap_t heap, int force);
```

Description

_udestroy destroys the *heap* you specify. It also returns the heap's memory to the system by calling the *release_fn* you supplied to _ucreate when you created the heap. If you did not supply a *release_fn*, _udestroy simply marks the heap as destroyed so no further operations can be performed. You must then return all the memory in the heap to the system.

Note: Whether or not you provide a *release_fn*, you must always return the initial block of memory (that you provided to _ucreate) to the system.

The *force* parameter controls the behavior of _udestroy if all allocated objects from the heap have not been freed. If you specify _FORCE for this parameter, _udestroy destroys the heap regardless of whether allocated objects remain in that process or in any other process that shares the heap. If you specify !_FORCE, the heap will not be destroyed if any objects are still allocated from it.

Typically, you call _uclose to close the heap before you destroy it. After you have destroyed a heap, any attempt to access it will have undefined results.

You cannot destroy *The Developer's Toolkit* run-time heap (_RUNTIME_HEAP).

Returns

_udestroy returns `0` whether the heap was destroyed or not. If the heap passed to it is not valid, _udestroy returns a nonzero value.

Example Code

The following example creates and opens a heap, performs operations on it, and then closes it. The program then calls _udestroy with the _FORCE parameter to force the destruction of the heap. _udestroy calls `release_fn` to return the memory to the system.

```
#define  INCL_DOSMEMMGR             /* Memory Manager values */
#include <os2.h>
#include <bsememf.h>                /* Get flags for memory management  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
   void *p;

   /* Round up to the next chunk size */
   *length = ((*length) / 65536) * 65536 + 65536;
   *clean = _BLOCK_CLEAN;
   DosAllocMem(&p, *length, PAG_COMMIT | PAG_READ | PAG_WRITE);
   return (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
   DosFreeMem(p);
   return;
}

int main(void)
{
   void    *initial_block;
   APIRET  rc;
   Heap_t  myheap;
   char    *ptr;

   /* Call DosAllocMem to get the initial block of memory */
   if (0 != (rc = DosAllocMem(&initial_block, 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   /* Create an expandable heap starting with the block declared earlier */
   if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                  _HEAP_REGULAR, get_fn, release_fn))) {
      puts("_ucreate failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != _uopen(myheap)) {
      puts("_uopen failed.");
      exit(EXIT_FAILURE);
   }

   /* Force user heap to grow */
   ptr = _umalloc(myheap, 100000);

   _uclose(myheap);

   if (0 != _udestroy(myheap, _FORCE)) {
      puts("_udestroy failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != (rc = DosFreeMem(initial_block))) {
      printf("DosFreeMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- _uaddmem
- _ucreate
- _uopen
- _uclose
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_uheapchk - Validate Memory Heap

_uheapchk - Validate Memory Heap

Syntax

```
#include <umalloc.h>
int _uheapchk(Heap_t heap);
```

Description

_uheapchk checks the *heap* you specify for minimal consistency by checking all allocated and freed objects on the heap.

_uheapchk works just like _heapchk, except that you specify the heap to check; _heapchk always checks the default heap.

Note: Using the _uheapchk, _uheapset, and _uheap_walk functions (and their equivalents for the default heap) may add overhead to each object allocated from the heap.

Returns

_uheapchk returns one of the following values, defined in both `<umalloc.h>` and `<malloc.h>`:

_HEAPBADBEGIN	The heap specifed is not valid. It may have been closed or destroyed.
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	The heap appears to be consistent.

Example Code

This example creates a heap and performs memory operations on it. It then calls _uheapchk to validate the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t  myheap;
   char     *ptr;
   int      rc;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = _ucalloc(myheap, 100, 1))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   *(ptr - 1) = 'x';     /* overwrite storage that was not allocated */

   if (_HEAPOK != (rc = _uheapchk(myheap))) {
      switch(rc) {
         case _HEAPEMPTY:
            puts("The heap has not been initialized.");
            break;
         case _HEAPBADNODE:
            puts("A memory node is corrupted or the heap is damaged.");
            break;
         case _HEAPBADBEGIN:
            puts("The heap specified is not valid.");
            break;
      }
      exit(rc);
   }
   free(ptr);
   return 0;

   /****************************************************************************
      The output should be similar to :

      A memory node is corrupted or the heap is damaged.
   ****************************************************************************/
}
```

Related Information

- _heapchk
- _heapmin
- _uheapset

- _uheap_walk
- "Managing Memory" in the *VisualAge C++ Programming Guide*
- "Debugging Your Heaps" in the *VisualAge C++ Programming Guide*

_uheapmin - Release Unused Memory in User Heap

_uheapmin - Release Unused Memory in User Heap

Syntax

```
#include <umalloc.h>
int _uheapmin(Heap_t heap);
```

Description

_uheapmin returns all unused memory blocks from the *heap* you specify to the operating system.

_uheapmin works just like _heapmin, except that you specify the heap to use; _heapmin always uses the default heap. A debug version of this function, _debug_uheapmin, is also provided.

To return the memory, _uheapmin calls the *release_fn* you supplied when you created the heap with _ucreate. If you did not supply a *release_fn*, _uheapmin has no effect and simply returns `0`.

Returns

If successful, _uheapmin returns `0`. A nonzero return value indicates failure. Passing _uheapmin a heap that is not valid has undefined results.

Example Code

The following example creates a heap and then allocates and frees a large block of memory from it. It then calls _uheapmin to return free blocks of memory to the system.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   /* Allocate a large object */
   if (NULL == (ptr = _umalloc(myheap, 60000))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 60000);
   free(ptr);

   /* _uheapmin will attempt to return the freed object to the system */
   if (0 != _uheapmin(myheap)) {
      puts("_uheapmin failed.");
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- _heapmin
- _ucreate
- Differentiating between Memory Management Functions
- "Managing Memory" in the *VisualAge C++ Programming Guide*
- "Debugging Your Heaps" in the *VisualAge C++ Programming Guide*

_uheapset - Validate and Set Memory Heap

_uheapset - Validate and Set Memory Heap

Syntax

```
#include <umalloc.h>
int _heapset(Heap_t heap, unsigned int fill);
```

Description

_uheapset checks the *heap* you specify for minimal consistency by checking all allocated and freed objects on the heap (similar to _uheapchk). It then sets each byte of the heap's free objects to the value of *fill*.

Using _uheapset can help you locate problems where your program continues to use a freed pointer to an object. After you set the free heap to a specific value, when your program tries to interpret the set values in the freed object as data, unexpected results occur, indicating a problem.

_uheapset works just like _heapset, except that you specify the heap to check; _heapset always checks the default heap.

Note: Using the _uheapchk, _uheapset, and _uheap_walk functions (and their equivalents for the default heap) may add overhead to each object allocated from the heap.

Returns

_uheapset returns one of the following values, defined in both `<umalloc.h>` and `<malloc.h>`:

_HEAPBADBEGIN	The heap specified is not valid. It may have been closed or destroyed.
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	The heap appears to be consistent.

Example Code

This example creates a heap and allocates and frees memory from it. It then calls _uheapset to set the freed memory to a value.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;
   int     rc;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = _umalloc(myheap, 100))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'A', 10);
   free(ptr);

   if (_HEAPOK != (rc = _uheapset(myheap, 'x'))) {
      switch(rc) {
         case _HEAPEMPTY:
            puts("The heap has not been initialized.");
            break;
         case _HEAPBADNODE:
            puts("A memory node is corrupted or the heap is damaged.");
            break;
         case _HEAPBADBEGIN:
            puts("The heap specified is not valid.");
            break;
      }
      exit(rc);
   }
   return 0;
}
```

Related Information

- _heapmin
- _heapset
- _uheapchk
- _uheap_walk
- "Managing Memory" in the *VisualAge C++ Programming Guide*
- "Debugging Your Heaps" in the *VisualAge C++ Programming Guide*

_uheap_walk - Return Information about Memory Heap

_uheap_walk - Return Information about Memory Heap

Syntax

```
#include <umalloc.h>
int _uheap_walk(Heap_t heap, int (*callback_fn)(const void *object,
                     size_t size, int flag, int status,
                     const char* file, int line) );
```

Description

_uheap_walk traverses the *heap* you specify, and, for each allocated or freed object, it calls the *callback_fn* function that you provide. _uheap_walk works just like _heap_walk, except that you specify the heap to be traversed; _heap_walk always traverses the default heap.

For each object, _uheap_walk passes your function:

object
A pointer to the object.

size
The size of the object.

flag
The value _USEDENTRY, if the object is currently allocated, or _FREEENTRY, if the object has been freed. (Both values are defined in `<malloc.h>`.)

status
One of the following values, defined in both `<umalloc.h>` and `<malloc.h>`, depending on the status of the object:

_HEAPBADBEGIN	The heap specified is not valid. It may have been closed or destroyed.
_HEAPBADNODE	A memory node is corrupted, or the heap is damaged.
_HEAPEMPTY	The heap has not been initialized.

_HEAPOK	The heap appears to be consistent.

file The name of the file where the object was allocated or freed.

line The line where the object was allocated or freed.

_uheap_walk provides information about all objects, regardless of which memory management functions were used to allocate them. However, the *file* and *line* information are only available if the object was allocated and freed using the debug versions of the memory management functions. Otherwise, *file* is NULL and *line* is `0`.

_uheap_walk calls *callback_fn* for each object until one of the following occurs:

- All objects have been traversed.
- *callback_fn* returns a nonzero value to _heap_walk.
- It cannot continue because of a problem with the heap.

You may want to code your *callback_fn* to return a nonzero value if the status of the object is not _HEAPOK. Even if *callback_fn* returns `0` for an object that is corrupted, _heap_walk cannot continue because of the state of the heap and returns to its caller.

You can use *callback_fn* to process information from _uheap_walk in various ways. For example, you may want to print the information to a file, or use it to generate your own error messages. You can use the information to look for memory leaks and objects incorrectly allocated or freed from the heap. It can be especially useful to call _uheap_walk when _uheapchk returns an error.

Note:

- Using the _uheapchk, _uheapset, and _uheap_walk functions (and their equivalents for the default heap) may add overhead to each object allocated from the heap

- _uheap_walk locks the heap while it traverses it, to ensure that no other operations use the heap until _uheap_walk finishes. As a result, in your *callback_fn* , you cannot call any critical functions in the run-time library, either explicitly or by calling another function that calls a critical function. See the *VisualAge C++ Programming Guide* for a list of critical functions.

Returns

_uheap_walk returns the last value of *status* to the caller.

Example Code

This example creates a heap and performs memory operations on it. _uheap_walk then traverses the heap and calls `callback_function` for each memory object. The `callback_function` prints a message about each memory block.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int callback_function(const void *pentry, size_t sz, int useflag, int status,
                      const char *filename, size_t line)
{
   if (_HEAPOK != status) {
      puts("status is not _HEAPOK.");
      exit(status);
   }
   if (_USEDENTRY == useflag)
      printf("allocated  %p     %u\n", pentry, sz);
   else
      printf("freed      %p     %u\n", pentry, sz);
   return 0;
}



int main(void)
{
   Heap_t  myheap;
   char    *p1, *p2, *p3;

   /* User default heap as user heap */
   myheap = _udefault(NULL);
   if (NULL == (p1 = _umalloc(myheap, 100)) ||
       NULL == (p2 = _umalloc(myheap, 200)) ||
       NULL == (p3 = _umalloc(myheap, 300))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   free(p2);
   puts("usage      address   size\n-----      -------   ----");

   _uheap_walk(myheap, callback_function);

   free(p1);
   free(p3);
   return 0;

   /****************************************************************************
      The output should be similar to :

      usage      address   size
      -----      -------   ----
      allocated  73A20     300
      allocated  738C0     100
       :
       :
      freed      73930     224
   ****************************************************************************/
}
```

Related Information

- _heapmin
- _heap_walk
- _uheapchk
- _uheapset
- "Managing Memory" in the *VisualAge C++ Programming Guide*
- "Debugging Your Heaps" in the *VisualAge C++ Programming Guide*

_ultoa - Convert Unsigned Long Integer to String

_ultoa - Convert Unsigned Long Integer to String

Syntax

```
#include <stdlib.h>
char *_ultoa(unsigned long value, char *string, int radix);
```

Description

_ultoa converts the digits of the given unsigned long *value* to a null-terminated character string and stores the result in *string*. No overflow checking is performed. The *radix* argument specifies the base of *value*; it must be in the range of 2 through 36.

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes, including the null character (\0).

Returns

_ultoa returns a pointer to *string*. There is no error return value.

Example Code

This example converts the digits of the value 255 to decimal, binary, and hexadecimal representations.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char buffer[10];
   char *p;

   p = _ultoa(255UL, buffer, 10);
   printf("The result of _ultoa(255) with radix of 10 is %s\n", p);
   p = _ultoa(255UL, buffer, 2);
   printf("The result of _ultoa(255) with radix of 2 is %s\n", p);
   p = _ultoa(255UL, buffer, 16);
   printf("The result of _ultoa(255) with radix of 16 is %s\n", p);
   return 0;

   /****************************************************************************
      The output should be:

      The result of _ultoa(255) with radix of 10 is 255
      The result of _ultoa(255) with radix of 2 is 11111111
      The result of _ultoa(255) with radix of 16 is ff
   ****************************************************************************/
}
```

Related Information

- _ecvt
- _fcvt
- _gcvt
- _itoa
- _ltoa

_umalloc - Reserve Memory Blocks from User Heap

_umalloc - Reserve Memory Blocks from User Heap

Syntax

```
#include <umalloc.h>
void *_umalloc(Heap_t heap, size_t size);
```

Description

_umalloc allocates a memory block of *size* bytes from the *heap* you specify. Unlike _ucalloc, _umalloc does not initialize all bits to 0.

_umalloc works just like malloc, except that you specify the heap to use; malloc always allocates from the default heap. A debug version of this function, _debug_umalloc, is also provided.

If the *heap* does not have enough memory for the request, _umalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To reallocate or free memory allocated with _umalloc, use the non-heap-specific realloc and free. These functions always check what heap the memory was allocated from.

Returns

_umalloc returns a pointer to the reserved space. If *size* was specified as 0, or if your *getmore_fn* cannot provide enough memory, _umalloc returns NULL. Passing _umalloc a heap that is not valid results in undefined behavior.

Example Code

This example creates a heap and uses _umalloc to allocate memory from the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = _umalloc(myheap, 100))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   free(ptr);
   return 0;
}
```

Related Information

- calloc
- free
- malloc
- realloc
- _ucalloc
- _ucreate
- "Managing Memory" in the *VisualAge C++ Programming Guide*

----

umask - Sets File Mask of Current Process

umask - Sets File Mask of Current Process

Syntax

```
#include <io.h>
#include <sys\stat.h>
int umask(int pmode);
```

Description

umask sets the file permission mask of the environment for the currently running process to the mode specified by *pmode* . The file permission mask modifies the permission setting of new files created by creat, open, or _sopen.

If a bit in the mask is 1, the corresponding bit in the requested permission value of the file is set to `0` (disallowed). If a bit in the mask is `0`, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The possible values for *pmode* are defined in `<sys\stat.h>`: compact break=fit.

Value	Meaning
S_IREAD	No effect
S_IWRITE	Writing not permitted
S_IREAD | S_IWRITE	Writing not permitted.

If the write bit is set in the mask, any new files will be read-only. You cannot give write-only permission, meaning that setting the read bit has no effect.

Returns

umask returns the previous value of *pmode* . A return value of `-1` indicates that the value used for *pmode* was not valid, and `errno` is set to EINVAL.

Example Code

This example sets the permission mask to create a write-only file.

```
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

int main(void)
{
   int oldMask;

   oldMask = umask(S_IWRITE);
   printf("\nDefault system startup mask is %d.\n", oldMask);
   return 0;

   /****************************************************************************
      The output should be:

      Default system startup mask is 0.
   ****************************************************************************/
}
```

Related Information

- chmod
- creat
- open
- _sopen

----

ungetc - Push Byte onto Input Stream

ungetc - Push Byte onto Input Stream

Syntax

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

`ungetc` pushes the byte *c* back onto the given input *stream* . However, only one sequential byte is guaranteed to be pushed back onto the input stream if you call `ungetc` consecutively. The *stream* must be open for reading. A subsequent read operation on the *stream* starts with *c* . The byte *c* cannot be the EOF character.

Bytes placed on the stream by `ungetc` will be erased if `fseek`, `fsetpos`, `rewind`, or `fflush` is called before the byte is read from the *stream* .

Returns

`ungetc` returns the integer argument *c* converted to an `unsigned char`, or `EOF` if *c* cannot be pushed back.

Example Code

In this example, the `while` statement reads decimal digits from an input data stream by using arithmetic statements to compose the numeric values of the numbers as it reads them. When a nondigit character appears before the end of the file, `ungetc` replaces it in the input stream so that later input functions can process it.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;
   unsigned int result = 0;

   while (EOF != (ch = getc(stdin)) && isdigit(ch))
      result = result * 10 + ch -'0';
   if (EOF != ch)
      /* Push back the nondigit character onto the input stream               */
      ungetc(ch, stdin);

   printf("Input number : %d\n", result);
   return 0;

   /****************************************************************************
      For the following input:

      12345s

      The output should be:

      Input number : 12345
   ****************************************************************************/
}
```

Related Information

- getc - getchar
- fflush
- fseek
- fsetpos
- putc - putchar
- rewind
- _ungetch

_ungetch - Push Character Back to Keyboard

_ungetch - Push Character Back to Keyboard

Syntax

```
#include <conio.h>
int _ungetch(int c);
```

Description

_ungetch pushes the character *c* back to the keyboard, causing *c* to be the next character read. _ungetch fails if called more than once before the next read operation. The character *c* cannot be the EOF character.

Returns

If successful, _ungetch returns the character *c*. A return value of EOF indicates an error.

Example Code

This example uses _getch to read a string delimited by the character 'x'. It then calls _ungetch to return the delimiter to the keyboard buffer. Other input routines can then process the delimiter.

```
#include <conio.h>
#include <stdio.h>

int main(void)
{
   int ch;

   printf("Type in some letters.\n");
   printf("If you type in an 'x', the program ends.\n");
   for (; ; ) {
      ch = _getch();
      if ('x' == ch) {
         _ungetch(ch);
         break;
      }
      _putch(ch);
   }
   ch = _getch();
   printf("\nThe last character was '%c'.", ch);
   return 0;

   /****************************************************************************
      Here is the output from a sample run:

      Type in some letters.
      If you type in an 'x', the program ends.
      One Two Three Four Five Si
      The last character was 'x'.
   ****************************************************************************/
}
```

Related Information

- _cscanf
- _getch - _getche
- _putch
- ungetc

----

ungetwc - Push Wide Character onto Input Stream

ungetwc - Push Wide Character onto Input Stream

Syntax

```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t wc, FILE *stream);
```

Description

ungetwc pushes the wide character by *wc* back onto the input *stream* . The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (on the *stream*) to a file positioning function (fseek, fsetpos, or rewind) discards any pushed-back wide characters for the stream.

The external storage corresponding to the stream is unchanged. There is always at least one wide character of push-back.

If the value of *wc* is WEOF, the operation fails and the input stream is unchanged.

A successful call to the ungetwc function clears the EOF indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.

For a text stream, the file position indicator is backed up by one wide character. This affects ftell, fflush, fseek (with SEEK_CUR), and fgetpos.

For a binary stream, the position indicator is unspecified until all characters are read or discarded, unless the last character is pushed back, in which case the file position indicator is backed up by one wide character. This affects ftell, fseek (with SEEK_CUR), fgetpos, and fflush.

After calling ungetwc, flush the buffer or reposition the stream pointer before calling a read function for the stream, unless EOF has been reached. After a read operation on the stream, flush the buffer or reposition the stream pointer before calling ungetwc.

Note:

- Only 1 wide character may be pushed back.
- The position on the stream after a successful call to ungetwc is one wide character prior to the current position.

Returns

ungetwc returns the wide character pushed back after conversion, or WEOF if the operation fails.

Example Code

This example reads in wide characters from `stream`, and then calls ungetwc to push the characters back to the `stream`.

```
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE         *stream;
   wint_t       wc;
   wint_t       wc2;
   unsigned int result = 0;

   if (NULL == (stream = fopen("ungetwc.dat", "r+"))) {
      printf("Unable to open file.\n");
      exit(EXIT_FAILURE);
   }

   while (WEOF != (wc = fgetwc(stream)) && iswdigit(wc))
      result = result * 10 + wc - L'0';

   if (WEOF != wc)
      ungetwc(wc, stream);    /* Push the nondigit wide character back */

   /* get the pushed back character */
   if (WEOF != (wc2 = fgetwc(stream))) {
      if (wc != wc2) {
         printf("Subsequent fgetwc does not get the pushed back character.\n");
         exit(EXIT_FAILURE);
      }
      printf("The digits read are '%i'\n"
             "The character being pushed back is '%lc'", result, wc2);
   }
   return 0;

   /****************************************************************************
      Assuming the file ungetwc.dat contains:

      12345ABCDE67890XYZ

      The output should be similar to :

      The digits read are '12345'
      The character being pushed back is 'A'
   ****************************************************************************/
}
```

Related Information

- fflush
- fseek
- fsetpos
- getwc
- putwc
- ungetc

----

unlink - Delete File

unlink - Delete File

Syntax

```
#include <stdio.h>  /* also in <io.h> */
int unlink(const char *pathname);
```

Description

unlink deletes the file specified by *pathname* .

Portability Note For portability, use the ANSI/ISO function remove instead of unlink.

Returns

unlink returns `0` if the file is successfully deleted. A return value of -1 indicates an error, and errno is set to one of the following values: compact break=fit.

Value	Meaning
EACCESS	The path name specifies a read-only file or a directory.
EISOPEN	The file is open.
ENOENT	An incorrect path name was specified, or the file or path name was not found.

Example Code

This example deletes the file `tmpfile` from the system or prints an error message if unable to delete it.

```
#include <stdio.h>

int main(void)
{
   if (-1 == unlink("tmpfile"))
      perror("Cannot delete tmpfile");
   else
      printf("tmpfile has been successfully deleted\n");
   return 0;

   /****************************************************************************
      If the file "tmpfile" exists, the output should be:

      tmpfile has been successfully deleted
   ****************************************************************************/
}
```

Related Information

- remove
- _rmtmp

_uopen - Open Heap for Use

_uopen - Open Heap for Use

Syntax

```
#include <umalloc.h>
int _uopen(Heap_t heap);
```

Description

_uopen allows the current process to use the *heap* you specify. If the heap is shared, you must call _uopen in each process that will allocate or free from the heap. See "Managing Memory" in the *VisualAge C++ Programming Guide* for more information about sharing heaps, and about creating and using heaps in general.

Returns

If successful, _uopen returns 0. A nonzero return code indicates failure. Passing _uopen a heap that is not valid results in undefined behavior.

Example Code

The following example creates a fixed-size heap, then uses _uopen to open it. The program then performs operations on the heap, and closes and destroys it.

```
#define  INCL_DOSMEMMGR             /* Memory Manager values */
#include <os2.h>
#include <bsememf.h>                /* Get flags for memory management  */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   void    *initial_block;
   APIRET  rc;
   Heap_t  myheap;
   char    *p;

   /* Call DosAllocMem to get the initial block of memory */
   if (0 != (rc = DosAllocMem(&initial_block, 65536,
                              PAG_WRITE | PAG_READ | PAG_COMMIT))) {
      printf("DosAllocMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   /* Create a fixed size heap starting with the block declared earlier */
   if (NULL == (myheap = _ucreate(initial_block, 65536, _BLOCK_CLEAN,
                                  _HEAP_REGULAR, NULL, NULL))) {
      puts("_ucreate failed.");
      exit(EXIT_FAILURE);
   }
   if (0 != _uopen(myheap)) {
      puts("_uopen failed.");
      exit(EXIT_FAILURE);
   }
   p = _umalloc(myheap, 100);
   free(p);
   if (0 != _uclose(myheap)) {
      puts("_uclose failed");
      exit(EXIT_FAILURE);
   }
   if (0 != (rc = DosFreeMem(initial_block))) {
      printf("DosFreeMem error: return code = %ld\n", rc);
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

Related Information

- _uaddmem
- _uclose
- _ucreate
- "Managing Memory" in the *VisualAge C++ Programming Guide*

_ustats - Get Information about Heap

_ustats - Get Information about Heap

Syntax

```
#include <umalloc.h>
int _ustats(Heap_t heap, _HEAPSTATS *hpinfo);
```

Description

_ustats gets information about the *heap* you specify and stores it in the *hpinfo* structure you pass to it.

The `_HEAPSTATS` structure type is defined in `<umalloc.h>`. The members it contains and the information that _ustats stores in each is as follows: compact break=fit.

`_provided` How much memory the heap holds (excluding memory used for overhead for the heap)

`_used` How much memory is currently allocated from the heap

`_shared` Whether the memory is shared (`_shared` is 1) or not (`_shared` is `0`)

`_maxfree` The size of the largest contiguous piece of memory available on the heap

Returns

If successful, _ustats returns `0`. A nonzero return code indicates failure. Passing _ustats a heap that is not valid results in undefined behavior.

Example Code

This example creates a heap and allocates memory from it. It then calls _ustats to print out information about the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

int main(void)
{
   Heap_t      myheap;
   _HEAPSTATS  myheap_stat;
   char        *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);
   if (NULL == (ptr = _umalloc(myheap, 100))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   if (0 != _ustats(myheap, &myheap_stat)) {
      puts("_ustats failed.");
      exit(EXIT_FAILURE);
   }
   printf ("_provided: %u\n", myheap_stat._provided);
   printf ("_used    : %u\n", myheap_stat._used);
   printf ("_shared  : %u\n", myheap_stat._shared);
   printf ("_max_free: %u\n", myheap_stat._max_free);
   free(ptr);
   return 0;
   /****************************************************************************
      The output should be similar to :

      _provided: 65264
      _used    : 14304
      _shared  : 0
      _max_free: 50960
   ****************************************************************************/
}
```

Related Information

- _mheap
- _ucreate
- "Managing Memory" in the *VisualAge C++ Programming Guide*

----

utime - Set Modification Time

utime - Set Modification Time

Syntax

```
#include <sys\utime.h>
#include <sys\types.h>
int utime(char *pathname, struct utimbuf *times);
```

Description

utime sets the modification time for the file specified by *pathname* . The process must have write access to the file; otherwise, the time cannot be changed.

Although the `utimbuf` structure contains a field for access time, only the modification time is set in the OS/2 operating system. If *times* is a `NULL` pointer, the modification time is set to the current time. Otherwise, *times* must point to a structure of type `utimbuf`, defined in `<sys\utime.h>`. The modification time is set from the `modtime` field in this structure.

utime accepts only even numbers of seconds. If you enter an odd number of seconds, the function rounds it down.

Returns

utime returns `0` if the file modification time was changed. A return value of -1 indicates an error, and errno is set to one of the following values: compact break=fit.

Value	Meaning
EACCESS	The path name specifies a directory or read-only file.
EMFILE	There are too many open files. You must open the file to change its modification time.

ENOENT	The file path name was not found, or the file name was incorrectly specified.

Example Code

This example sets the last modification time of file `utime.dat` to the current time. It prints an error message if it cannot.

```
#include <sys\types.h>
#include <sys\utime.h>
#include <sys\stat.h>
#include <stdio.h>
#include <stdlib.h>

#define  FILENAME      "utime.dat"

int main(void)
{
   struct utimbuf ubuf;
   struct stat statbuf;
   FILE *fp;                                             /* File pointer     */

   /* creating file, whose date will be changed by calling utime              */
   fp = fopen(FILENAME, "w");

   /* write Hello World in the file                                           */
   fprintf(fp, "Hello World\n");

   /* close file                                                              */
   fclose(fp);

   /* seconds to current date from 1970 Jan 1                                 */
   /* Fri Dec 31 23:59:58 1999                                                */
   ubuf.modtime = 946702799;

   /* changing file modification time                                         */
   if (-1 == utime(FILENAME, &ubuf)) {
      perror("utime failed");
      remove(FILENAME);
      return EXIT_FAILURE;
   }

   /* display the modification time                                           */
   if (0 == stat(FILENAME, &statbuf))
      printf("The file modification time is %s", ctime(&statbuf.st_mtime));
   else
      printf("File could not be found\n");
   remove(FILENAME);
   return 0;

   /****************************************************************************
      The output should be:

      The file modification time is Fri Dec 31 23:59:58 1999
   ****************************************************************************/
}
```

Related Information

- fstat
- stat

----

va_arg - va_end - va_start - Access Function Arguments

va_arg - va_end - va_start - Access Function Arguments

Syntax

```
#include <stdarg.h>
var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);
```

Description

`va_arg`, `va_end`, and `va_start` access the arguments to a function when it takes a fixed number of required arguments and a variable number of optional arguments. All three of these are macros. You declare required arguments as ordinary parameters to the function and access the arguments through the parameter names.

`va_start` initializes the *arg_ptr* pointer for subsequent calls to `va_arg` and `va_end`.

The argument *variable_name* is the identifier of the rightmost named parameter in the parameter list (preceding `,` ...). Use `va_start` before `va_arg`. Corresponding `va_start` and `va_end` macros must be in the same function.

`va_arg` retrieves a value of the given *var_type* from the location given by *arg_ptr*, and increases *arg_ptr* to point to the next argument in the list. `va_arg` can retrieve arguments from the list any number of times within the function. The *var_type* argument must be one of `int`, `long`, `double`, `struct`, `union`, or pointer, or a `typedef` of one of these types.

`va_end` is needed to indicate the end of parameter scanning.

Returns

`va_arg` returns the current argument. `va_end` and `va_start` do not return a value.

Example Code

This example passes a variable number of arguments to a function, stores each argument in an array, and prints each argument.

```
#include <stdio.h>
#include <stdarg.h>

int vout(int max,...);

int main(void)
{
   vout(3, "Sat", "Sun", "Mon");
   printf("\n");
   vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
   return 0;
}

int vout(int max,...)
{
   va_list arg_ptr;
   int args = 0;
   char *days[7];

   va_start(arg_ptr, max);
   while (args < max) {
      days[args] = va_arg(arg_ptr, char *);
      printf("Day:  %s  \n", days[args++]);
   }
   va_end(arg_ptr);

   /****************************************************************************
      The output should be:

      Day:  Sat
      Day:  Sun
      Day:  Mon

      Day:  Mon
      Day:  Tues
      Day:  Wed
      Day:  Thurs
      Day:  Fri
   ****************************************************************************/
}
```

Related Information

- vfprintf
- vprintf
- vsprintf

----

vfprintf - Print Argument Data to Stream

vfprintf - Print Argument Data to Stream

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

Description

`vfprintf` formats and writes a series of characters and values to the output *stream*. `vfprintf` works just like `fprintf`, except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, `fprintf` can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

`vfprintf` converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for `printf`. For a description of the format string, see printf.

In extended mode, `vfprintf` also converts floating-point values of NaN and infinity to the strings `"NAN"` or `"nan"` and `"INFINITY"` or `"infinity"`. The case and sign of the string is determined by the format specifiers. See Infinity and NaN Support for more information on infinity and NaN values.

Returns

If successful, `vfprintf` returns the number of bytes written to *stream*. If an error occurs, the function returns a negative value.

Example Code

This example prints out a variable number of strings to the file `vfprintf.out`.

```
#include <stdarg.h>
#include <stdio.h>

#define FILENAME "vfprintf.o"

void vout(FILE *stream, char *fmt,...);

char fmt1[] = "%s  %s  %s  %s\n";

int main(void)
{
   FILE *stream;

   stream = fopen(FILENAME, "w");
   vout(stream, fmt1, "Sat", "Sun", "Mon", "Tue");
   fclose(stream);
   return 0;
   /*****************************************************************************
      After running the program, the output file should contain:

      Sat  Sun  Mon  Tue
   *****************************************************************************/
}

void vout(FILE *stream, char *fmt,...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vfprintf(stream, fmt, arg_ptr);
   va_end(arg_ptr);
}
```

Related Information

- fprintf
- printf
- va_arg - va_end - va_start
- vprintf
- vsprintf
- Infinity and NaN Support

----

vprintf - Print Argument Data

vprintf - Print Argument Data

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);
```

Description

`vprintf` formats and prints a series of characters and values to stdout. `vprintf` works just like `printf`, except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, `printf` can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

`vprintf` converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for `printf`. For a description of the format string, see printf.

In extended mode, `vprintf` also converts floating-point values of NaN and infinity to the strings `"NAN"` or `"nan"` and `"INFINITY"` or `"infinity"`. The case and sign of the string is determined by the format specifiers. See Infinity and NaN Support for more information on infinity and NaN values.

Returns

If successful, `vprintf` returns the number of bytes written to stdout. If an error occurs, `vprintf` returns a negative value.

Example Code

This example prints out a variable number of strings to stdout.

```
#include <stdarg.h>
```

```
#include <stdio.h>

void vout(char *fmt, ...);

int main(void)
{
   char fmt1[] = "%s  %s  %s\n";
   vout(fmt1, "Sat", "Sun", "Mon");
   return 0;

   /****************************************************************************
      The output should be:

      Sat  Sun  Mon
   ****************************************************************************/
}

void vout(char *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vprintf(fmt, arg_ptr);
   va_end(arg_ptr);
}
```

Related Information

- printf
- va_arg - va_end - va_start
- vfprintf
- vsprintf

----

vsprintf - Print Argument Data to Buffer

vsprintf - Print Argument Data to Buffer

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *target-string, const char *format, va_list arg_ptr);
```

Description

`vsprintf` formats and stores a series of characters and values in the buffer *target-string*. `vsprintf` works just like `sprintf`, except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, `sprintf` can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

`vsprintf` converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for `printf`. For a description of the format string, see printf.

In extended mode, `vsprintf` also converts floating-point values of NaN and infinity to the strings `"NAN"` or `"nan"` and `"INFINITY"` or `"infinity"`. The case and sign of the string is determined by the format specifiers. See Infinity and NaN Support for more information on infinity and NaN values.

Returns

If successful, `vsprintf` returns the number of bytes written to *target-string*. If an error occurs, `vsprintf` returns a negative value.

Example Code

This example assigns a variable number of strings to `string` and prints the resultant string.

```
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt,...);

char fmt1[] = "%s  %s  %s\n";

int main(void)
{
   char string[100];

   vout(string, fmt1, "Sat", "Sun", "Mon");
   printf("The string is: %s", string);
   return 0;

   /****************************************************************************
      The output should be:

      The string is: Sat  Sun  Mon
   ****************************************************************************/
}

void vout(char *string, char *fmt,...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vsprintf(string, fmt, arg_ptr);
   va_end(arg_ptr);
}
```

Related Information

- printf
- sprintf
- va_arg - va_end - va_start
- vfprintf
- vprintf
- Infinity and NaN Support

----

wait - Wait for Child Process

wait - Wait for Child Process

Syntax

```
#include <process.h>
int wait (int *stat_loc);
```

Description

wait delays a parent process until one of the immediate child processes stops. If all the child processes stop before wait is called, control returns immediately to the parent function.

If *stat_loc* is `NULL`, wait does not use it. If it is not `NULL`, wait places information about the return status and the return code ot the child process that ended in the location to which *stat_loc* points.

If the child process ended normally, with a call to the OS/2 DosExit function, the lowest-order byte of *stat_loc* is `0`, and the next higher-order byte contains the lowest-order byte of the argument passed to DosExit by the child process. The value of this byte depends on how the child process caused the system to call DosExit.

If the child process called exit, _exit, or `return` from `main`, the byte contains the lowest-order byte of the argument the child process passed to exit, _exit, or **return**. The value of the byte is undefined if the child process caused a DosExit call simply by reaching the end of `main`.

If the child process ended abnormally, the lowest-order byte of *stat_loc* contains the return status code from the OS/2 DosWaitChild function, and the next higher-order byte is `0`. See the *Control Program Guide and Reference* for details about DosWaitChild return status codes.

Returns

If wait returns after a normal end of a child process, it returns the process identifier of the child process to the parent process. A return value of -1 indicates an error, and errno is set to one of the following values: compact break=fit.

Value	Meaning
ECHILD	There were no child processes or they all ended before the call to wait. This value indicates that no child processes exist for the particular process.
EINTR	A child process ended unexpectedly.

Example Code

This example creates a new process called `CHILD.EXE`, specifying P_NOWAIT when the child process is called. The parent process calls wait and waits for the child process to stop running. The parent process then displays the return information of the child process in hexadecimal.

```
#include <stdio.h>
#include <process.h>

int stat_child;

int main(void)
{
   int pid;

   _spawnl(P_NOWAIT, "child2.exe", "child2.exe", NULL);
   if (-1 == (pid = wait(&stat_child)))
      perror("error in _spawnl"); /* display error status message             */
   else
      printf("child process %d just ran.\n", pid);

   printf("return information was 0x%X\n", stat_child);
   return 0;

   /****************************************************************************
      If the source for child2.exe is:

      #include <stdio.h>
      int main(void)
      {
          puts("child2.exe is an executable file");
          return 0;
      }

      Then the output should be similar to:

      child2.exe is an executable file
      child process 2423 just ran.
      return information was 0x0
   ****************************************************************************/
}
```

Related Information

- _cwait
- execl - _execvpe
- exit
- _exit
- _spawnl - _spawnvpe

----

wcscat - Concatenate Wide-Character Strings

wcscat - Concatenate Wide-Character Strings

Syntax

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);
```

Description

`wcscat` appends a copy of the string pointed to by *string2* to the end of the string pointed to by *string1* .

`wcscat` operates on null-terminated `wchar_t` strings. The string arguments to this function should contain a `wchar_t` null character marking the end of the string. Boundary checking is not performed.

Returns

`wcscat` returns a pointer to the concatenated *string1* .

Example Code

This example creates the wide character string `"computer program"` using `wcscat`.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t buffer1[SIZE] = L"computer";
   wchar_t *string = L" program";
   wchar_t *ptr;

   ptr = wcscat(buffer1, string);
   printf("buffer1 = %ls\n", buffer1);
   return 0;

   /****************************************************************************
      The output should be:

      buffer1 = computer program
   ****************************************************************************/
}
```

Related Information

- strcat
- strncat
- wcsncat

----

wcschr - Search for Wide Character

wcschr - Search for Wide Character

Syntax

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *string, wchar_t character);
```

Description

`wcschr` searches the wide-character *string* for the occurrence of the wide *character*. The wide *character* can be a `wchar_t` null character (`\0`); the `wchar_t` null character at the end of *string* is included in the search.

`wcschr` operates on null-terminated `wchar_t` strings. The string argument to this function should contain a `wchar_t` null character marking the end of the string.

Returns

`wcschr` returns a pointer to the first occurrence of *character* in *string*. If the character is not found, a `NULL` pointer is returned.

Example Code

This example finds the first occurrence of the character `p` in the wide-character string `"computer program"`.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t buffer1[SIZE] = L"computer program";
   wchar_t *ptr;
   wchar_t ch = L'p';

   ptr = wcschr(buffer1, ch);
   printf("The first occurrence of %lc in '%ls' is '%ls'\n", ch, buffer1, ptr);
   return 0;

   /****************************************************************************
      The output should be:

      The first occurrence of p in 'computer program' is 'puter program'
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- strspn
- wcscspn
- wcspbrk
- wcsrchr
- wcsspn
- wcswcs

----

wcscmp - Compare Wide-Character Strings

wcscmp - Compare Wide-Character Strings

Syntax

```
#include <wchar.h>
int wcscmp(const wchar_t *string1, const wchar_t *string2);
```

Description

`wcscmp` compares two wide-character strings.

`wcscmp` operates on null-terminated `wchar_t` strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string. Boundary checking is not performed when a string is added to or copied.

Returns

`wcscmp` returns a value indicating the relationship between the two strings, as follows: compact break=fit.

Value	**Meaning**
Less than `0`	*string1* less than *string2*
`0`	*string1* identical to *string2*
Greater than `0`	*string1* greater than *string2* .

Example Code

This example compares the wide-character string `string1` to `string2` using `wcscmp`.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   int result;
   wchar_t string1[] = L"abcdef";
   wchar_t string2[] = L"abcdefg";

   result = wcscmp(string1, string2);
   if (0 == result)
      printf("\"%ls\" is identical to \"%ls\"\n", string1, string2);
   else
      if (result < 0)
         printf("\"%ls\" is less than \"%ls\"\n", string1, string2);
      else
         printf("\"%ls\" is greater than \"%ls\"\n", string1, string2);
   return 0;

   /****************************************************************************
      The output should be:

      "abcdef" is less than "abcdefg"
   ****************************************************************************/
}
```

Related Information

- strcmp
- strcmpi
- stricmp
- strnicmp
- wcsncmp

wcscoll - Compare Wide-Character Strings

wcscoll - Compare Wide-Character Strings

Syntax

```
#include <wchar.h>
int wcscoll(const wchar_t *wcstr1, const wchar_t *wcstr2);
```

Description

wcscoll compares the wide-character string pointed to by *wcstr1* to the wide-character string pointed to by *wcstr2*, both interpreted according to the information in the LC_COLLATE category of the current locale.

wcscoll differs from the wcscmp function. wcscoll performs a comparison between two wide-character strings based on language collation rules as controlled by the LC_COLLATE category. In contrast, wcscmp performs a wide-character code to wide-character code comparison. If a string will be collated many times, as when inserting an entry into a sorted list, wcsxfrm followed by wcscmp can be more efficient.

Returns

wcscoll returns an integer value indicating the relationship between the strings, as listed below: compact break=fit.

Value	**Meaning**
Less than 0	*wcstr1* less than *wcstr2*
0	*wcstr1* equivalent to *wcstr2*
Greater than 0	*wcstr1* greater than *wcstr2*

If *wcs1* or *wcs2* contain characters outside the domain of the collating sequence, wcscoll sets errno to EILSEQ. If an error occurs, wcscoll sets errno to a nonzero value. There is no error return value.

Example Code

This example uses wcscoll to compare two wide-character strings.

```
#include <wchar.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   wchar_t *wcs1 = L"A wide string";
   wchar_t *wcs2 = L"a wide string";
   int     result;

   if (NULL == setlocale(LC_ALL, "Ja_JP")) {
      printf("setlocale failed.\n");
      exit(EXIT_FAILURE);
   }
   result = wcscoll(wcs1, wcs2);
   if (0 == result)
      printf("\"%ls\" is identical to \"%ls\"\n", wcs1, wcs2);
   else if (0 > result)
      printf("\"%ls\" is less than \"%ls\"\n", wcs1, wcs2);
   else
      printf("\"%ls\" is greater than \"%ls\"\n", wcs1, wcs2);
   return 0;

   /****************************************************************************
      The output should be similar to :

      "A wide string" is identical to "a wide string"
   ****************************************************************************/
}
```

Related Information

- strcoll
- setlocale
- wcscmp
- wcsxfrm

----

wcscpy - Copy Wide-Character Strings

wcscpy - Copy Wide-Character Strings

Syntax

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);
```

Description

`wcscpy` copies the contents of *string2* (including the ending `wchar_t` null character) into *string1* .

`wcscpy` operates on null-terminated `wchar_t` strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string. Boundary checking is not performed.

Returns

`wcscpy` returns a pointer to *string1* .

Example Code

This example copies the contents of `source` to `destination`.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t source[SIZE] = L"This is the source string";
   wchar_t destination[SIZE] = L"And this is the destination string";
   wchar_t *return_string;

   printf("destination is originally = \"%ls\"\n", destination);
   return_string = wcscpy(destination, source);
   printf("After wcscpy, destination becomes \"%ls\"\n", return_string);
   return 0;

   /****************************************************************************
      The output should be:

      destination is originally = "And this is the destination string"
      After wcscpy, destination becomes "This is the source string"
   ****************************************************************************/
}
```

Related Information

- strcpy
- strdup
- strncpy
- wcsncpy
- wcsxfrm

----

wcscspn - Find Offset of First Wide-Character Match

wcscspn - Find Offset of First Wide-Character Match

Syntax

```
#include <wchar.h>
size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

Description

`wcscspn` determines the number of `wchar_t` characters in the initial segment of the string pointed to by *string1* that do not appear in the string pointed to by *string2* .

`wcscspn` operates on null-terminated `wchar_t` strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

Returns

`wcscspn` returns the number of `wchar_t` characters in the segment.

Example Code

This example uses `wcscspn` to find the first occurrence of any of the characters `a`, `x`, `l`, or `e` in `string`.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t string[SIZE] = L"This is the source string";
   wchar_t *substring = L"axle";

   printf("The first %i characters in the string \"%ls\" are not in the "
      "string \"%ls\" \n", wcscspn(string, substring), string, substring);
   return 0;

   /****************************************************************************
      The output should be:

      The first 10 characters in the string "This is the source string" are
      not in the string "axle"?
   ****************************************************************************/
}
```

Related Information

- strcspn
- strspn
- wcsspn
- wcswcs

----

wcsftime - Convert to Formatted Date and Time

wcsftime - Convert to Formatted Date and Time

Syntax

```
#include <wchar.h>
size_t wcsftime(wchar_t *wdest, size_t maxsize,
                const wchar_t *format, const struct tm *timeptr);
```

Description

wcsftime converts the time and date specification in the *timeptr* structure into a wide-character string. It then stores the null-terminated string in the array pointed to by *wdest* according to the format string pointed to by *format* . *maxsize* specifies the maximum number of wide characters that can be copied into the array.

wcsftime works just like strftime, except that it uses wide characters.

The format string is a wide character string containing:

- Conversion specification characters.
- Ordinary wide characters, which are copied into the array unchanged.

The characters that are converted are determined by the LC_TIME category of the current locale and by the values in the time structure pointed to by *timeptr* . The time structure pointed to by *timeptr* is usually obtained by calling the gmtime or localtime function.

For details on the conversion specifiers you can use in the format string, see strftime.

Returns

If the total number of wide characters in the resulting string, including the terminating null wide character, does not exceed *maxsize* , wcsftime returns the number of wide characters placed into the array pointed to by *wdest* , not including the terminating null wide character; otherwise, wcsftime returns `0` and the contents of the array are indeterminate.

Example Code

This example obtains the date and time using localtime, formats the information with wcsftime, and prints the date and time.

```
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
   struct tm *timeptr;
   wchar_t   dest[100];
   time_t    temp;
   size_t    rc;

   temp = time(NULL);
   timeptr = localtime(&temp);
   rc = wcsftime(dest, sizeof(dest)-1,L" Today is %A,"
                L" %b %d.\n Time: %I:%M %p", timeptr);
   printf("%d characters placed in string to make:\n\n%ls\n", rc, dest);
   return 0;

   /****************************************************************************
      The output should be similar to :

      43 characters placed in string to make:
       Today is Thursday, Nov 10.
       Time: 04:56 PM
   ****************************************************************************/
}
```

Related Information

- gmtime
- localtime
- strftime
- strptime

----

wcslen - Calculate Length of Wide-Character String

wcslen - Calculate Length of Wide-Character String

Syntax

```
#include <wchar.h>
size_t wcslen(const wchar_t *string);
```

Description

`wcslen` computes the number of wide characters in the string pointed to by *string* .

Returns

`wcslen` returns the number of wide characters in *string* , excluding the terminating `wchar_t` null character.

Example Code

This example computes the length of the wide-character string `string`.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *string = L"abcdef";

   printf("Length of \"%ls\" is %i\n", string, wcslen(string));
   return 0;

   /****************************************************************************
      The output should be:

      Length of "abcdef" is 6
   ****************************************************************************/
}
```

Related Information

- mblen
- strlen

----

wcsncat - Concatenate Wide-Character Strings

wcsncat - Concatenate Wide-Character Strings

Syntax

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

Description

`wcsncat` appends up to *count* wide characters from *string2* to the end of *string1* , and appends a `wchar_t` null character to the result.

`wcsncat` operates on null-terminated wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

Returns

`wcsncat` returns *string1* .

Example Code

This example demonstrates the difference between `wcscat` and `wcsncat`. `wcscat` appends the entire second string to the first; `wcsncat` appends only the specified number of characters in the second string to the first.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t buffer1[SIZE] = L"computer";
   wchar_t *ptr;

   /* Call wcscat with buffer1 and " program"                           */

   ptr = wcscat(buffer1, L" program");
   printf("wcscat : buffer1 = \"%ls\"\n", buffer1);

   /* Reset buffer1 to contain just the string "computer" again         */

   memset(buffer1, L'\0', sizeof(buffer1));
   ptr = wcscpy(buffer1, L"computer");

   /* Call wcsncat with buffer1 and " program"                          */

   ptr = wcsncat(buffer1, L" program", 3);
   printf("wcsncat: buffer1 = \"%ls\"\n", buffer1);
   return 0;

   /****************************************************************************
      The output should be:

      wcscat : buffer1 = "computer program"
      wcsncat: buffer1 = "computer pr"
   ****************************************************************************/
}
```

Related Information

- strncat
- strcat
- wcscat
- wcsncmp
- wcsncpy

----

wcsncmp - Compare Wide-Character Strings

wcsncmp - Compare Wide-Character Strings

Syntax

```
#include <wchar.h>
int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

Description

`wcsncmp` compares up to *count* wide characters in *string1* to *string2* .

`wcsncmp` operates on null-terminated wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

Returns

`wcsncmp` returns a value indicating the relationship between the two strings, as follows: compact break=fit.

Value	Meaning
Less than `0`	*string1* less than *string2*
`0`	*string1* identical to *string2*
Greater than `0`	*string1* greater than *string2* .

Example Code

This example demonstrates the difference between `wcscmp`, which compares the entire strings, and `wcsncmp`, which compares only a specified number of wide characters in the strings.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          10

int main(void)
{
   int result;
   int index = 3;
   wchar_t buffer1[SIZE] = L"abcdefg";
   wchar_t buffer2[SIZE] = L"abcfg";
   void print_result(int, wchar_t *, wchar_t *);

   result = wcscmp(buffer1, buffer2);
   printf("Comparison of each character\n");
   printf("  wcscmp: ");
   print_result(result, buffer1, buffer2);
   result = wcsncmp(buffer1, buffer2, index);
   printf("\nComparison of only the first %i characters\n", index);
   printf("  wcsncmp: ");
   print_result(result, buffer1, buffer2);
   return 0;

   /****************************************************************************
      The output should be:

      Comparison of each character
        wcscmp: "abcdefg" is less than "abcfg"

      Comparison of only the first 3 characters
        wcsncmp: "abcdefg" is identical to "abcfg"
   *****************************************************************************/
}

void print_result(int res,wchar_t *p_buffer1,wchar_t *p_buffer2)
{
   if (0 == res)
      printf("\"%ls\" is identical to \"%ls\"\n", p_buffer1, p_buffer2);
   else
      if (res < 0)
         printf("\"%ls\" is less than \"%ls\"\n", p_buffer1, p_buffer2);
      else
         printf("\"%ls\" is greater than \"%ls\"\n", p_buffer1, p_buffer2);
}
```

Related Information

- strncmp
- strcmp
- strcoll
- strcmpi
- stricmp
- strnicmp
- wcscmp
- wcsncat
- wcsncpy

----

wcsncpy - Copy Wide-Character Strings

wcsncpy - Copy Wide-Character Strings

Syntax

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);
```

Description

`wcsncpy` copies up to *count* wide characters from *string2* to *string1* . If *string2* is shorter than *count* characters, *string1* is padded out to *count* characters with `wchar_t` null characters.

`wcsncpy` operates on null-terminated wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

Returns

`wcsncpy` returns a pointer to *string1* .

Example Code

This example demonstrates the difference between `wcscpy`, which copies the entire wide-character string, and `wcsncpy`, which copies a specified number of wide characters from the string.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t source[SIZE] = L"123456789";
   wchar_t source1[SIZE] = L"123456789";
   wchar_t destination[SIZE] = L"abcdefg";
   wchar_t destination1[SIZE] = L"abcdefg";
   wchar_t *return_string;
   int index = 5;

   /* This is how wcscpy works                                                */

   printf("destination is originally = '%ls'\n", destination);
   return_string = wcscpy(destination, source);
   printf("After wcscpy, destination becomes '%ls'\n\n", return_string);

   /* This is how wcsncpy works                                               */

   printf("destination1 is originally = '%ls'\n", destination1);
   return_string = wcsncpy(destination1, source1, index);
   printf("After wcsncpy, destination1 becomes '%ls'\n", return_string);
   return 0;

   /****************************************************************************
      The output should be:

      destination is originally = 'abcdefg'
      After wcscpy, destination becomes '123456789'

      destination1 is originally = 'abcdefg'
      After wcsncpy, destination1 becomes '12345fg'
   ****************************************************************************/
}
```

Related Information

- strcpy
- strncpy
- wcscpy
- wcsncat
- wcsncmp

----

wcspbrk - Locate Wide Characters in String

wcspbrk - Locate Wide Characters in String

Syntax

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);
```

Description

`wcspbrk` locates the first occurrence in the string pointed to by *string1* of any wide character from the string pointed to by *string2*.

Returns

`wcspbrk` returns a pointer to the character. If *string1* and *string2* have no wide characters in common, `wcspbrk` returns `NULL`.

Example Code

This example uses `wcspbrk` to find the first occurrence of either `a` or `b` in the array *string*.

```
#include <stdio.h>
```

```
#include <wchar.h>

int main(void)
{
   wchar_t *result;
   wchar_t *string = L"A Blue Danube";
   wchar_t *chars = L"ab";

   result = wcspbrk(string, chars);
   printf("The first occurrence of any of the characters \"%ls\" in "
      "\"%ls\" is \"%ls\"\n", chars, string, result);
   return 0;

   /****************************************************************************
      The output should be similar to:

      The first occurrence of any of the characters "ab" in "A Blue Danube"
      is "anube"
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- strspn
- wcschr
- wcscmp
- wcscspn
- wcsncmp
- wcsrchr
- wcsspn
- wcswcs

----

wcsrchr - Locate Wide Character in String

wcsrchr - Locate Wide Character in String

Syntax

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *string, wint_t character);
```

Description

`wcsrchr` locates the last occurrence of *character* in the string pointed to by *string*. The terminating `wchar_t` null character is considered to be part of the string.

Returns

`wcsrchr` returns a pointer to the character, or a `NULL` pointer if *character* does not occur in the string.

Example Code

This example compares the use of `wcschr` and `wcsrchr`. It searches the string for the first and last occurrence of `p` in the wide character string.

```
#include <stdio.h>
#include <wchar.h>
```

```
#define  SIZE         40

int main(void)
{
   wchar_t buf[SIZE] = L"computer program";
   wchar_t *ptr;
   int ch = 'p';

   /* This illustrates wcschr                                            */

   ptr = wcschr(buf, ch);
   printf("The first occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr);

   /* This illustrates wscrchr                                           */

   ptr = wcsrchr(buf, ch);
   printf("The last occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr);
   return 0;

   /****************************************************************************
      The output should be:

      The first occurrence of p in 'computer program' is 'puter program'
      The last occurrence of p in 'computer program' is 'program'
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- strspn
- wcschr
- wcscspn
- wcsspn
- wcswcs
- wcspbrk

----

wcsspn - Search Wide-Character Strings

wcsspn - Search Wide-Character Strings

Syntax

```
#include <wchar.h>
size_t wcsspn(const wchar_t *string1, const wchar_t *string2);
```

Description

`wcsspn` scans *string1* for the wide characters contained in *string2*. It stops when it encounters a character in *string1* that is not in *string2*.

Returns

`wcsspn` returns the number of wide characters from *string1* that are found in *string2*.

Example Code

This example finds the first occurrence in the array `string` of a wide character that is not an a, b, or c. Because the string in this example is `cabbage`, `wcsspn` returns `5`, the index of the segment of `cabbage` before a character that is not an `a`, `b`, or `c`.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *string = L"cabbage";
   wchar_t *source = L"abc";
   int index;

   index = wcsspn(string, L"abc");
   printf("The first %d characters of \"%ls\" are found in \"%ls\"\n", index,
      string, source);
   return 0;

   /****************************************************************************
      The output should be:

      The first 5 characters of "cabbage" are found in "abc"
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- strspn
- wcschr
- wcscspn
- wcsrchr
- wcsspn
- wcswcs
- wcspbrk

----

wcsstr - Locate Wide-Character Substring

wcsstr - Locate Wide-Character Substring

Syntax

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

Description

wcsstr locates the first occurrence of *wcs2* in *wcs1*. In the matching process, wcsstr ignores the wchar_t null character that ends *wcs2*.

The behavior of wcsstr is affected by the LC_CTYPE category of the current locale.

Returns

wcsstr returns a pointer to the beginning of the first occurrence of *wcs2* in *wcs1*. If *wcs2* does not appear in *wcs1*, wcsstr returns `NULL`. If *wcs2* points to a wide-character string with zero length, wcsstr returns *wcs1*.

Example Code

This example uses wcsstr to find the first occurrence of `hay` in the wide-character string `needle in a haystack`.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs1 = L"needle in a haystack";
   wchar_t *wcs2 = L"hay";

   printf("result: \"%ls\"\n", wcsstr(wcs1, wcs2));
   return 0;

   /****************************************************************************
      The output should be similar to :

      result: "haystack"
   ****************************************************************************/
}
```

Related Information

- strstr
- wcschr
- wcsrchr
- wcswcs

----

wcstod - Convert Wide-Character String to Double

wcstod - Convert Wide-Character String to Double

Syntax

```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

Description

wcstod converts the wide-character string pointed to by *nptr* to a double value. The *nptr* parameter points to a sequence of characters that can be interpreted as a numerical value of type `double`. wcstod stops reading the string at the first character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string.

wcstod expects *nptr* to point to a string with the following form:

```
>>                                                         >
      white-space     +     digits
                                      .     digits
                              .  digits

>                            ><
      e           digits
      E     +
```

Note: The character used for the decimal point (shown as `.` in the above diagram) depends on the LC_NUMERIC category of the current locale.

The white space character is determined by the LC_CTYPE category of the current locale.

Returns

wcstod function returns the converted double value. If no conversion could be performed, wcstod returns `0`. If the correct value is outside the range of representable values, wcstod returns +HUGE_VAL or -HUGE_VAL (according to the sign of the value), and sets `errno` to ERANGE. If the correct value would cause underflow, wcstod returns `0` and sets `errno` to ERANGE.

If the string *nptr* points to is empty or does not have the expected form, no conversion is performed, and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Example Code

This example uses wcstod to convert the string `wcs` to a floating-point value.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"3.1415926This stopped it";
   wchar_t *stopwcs;

   printf("wcs = \"%ls\"\n", wcs);
   printf("   wcstod = %f\n", wcstod(wcs, &stopwcs));
   printf("   Stop scanning at \"%ls\"\n", stopwcs);
   return 0;

   /****************************************************************************
      The output should be similar to :

      wcs = "3.1415926This stopped it"
         wcstod = 3.141593
         Stop scanning at "This stopped it"
   ****************************************************************************/
}
```

Related Information

- strtod
- strtol
- strtold
- strtoul
- wcstol
- wcstoul

----

wcstok - Tokenize Wide-Character String

wcstok - Tokenize Wide-Character String

Syntax

```
#include <wchar.h>
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);
```

Description

wcstok reads *wcs1* as a series of zero or more tokens and *wcs2* as the set of wide characters serving as delimiters for the tokens in *wcs1* . A sequence of calls to wcstok locates the tokens inside *wcs1* . The tokens can be separated by one or more of the delimiters from *wcs2* . The third argument points to a wide-character pointer that you provide, where wcstok stores information necessary for it to continue scanning the same string.

When wcstok is first called for the wide-character string *wcs1* , it searches for the first token in *wcs1* , skipping over leading delimiters. wcstok returns a pointer to the first token.

To read the next token from *wcs1* , call wcstok with `NULL` as the first parameter (*wcs1*). This `NULL` parameter causes wcstok to search for the next token in the previous token string. Each delimiter is replaced by a null character to terminate the token.

wcstok always stores enough information in the pointer *ptr* so that subsequent calls, with `NULL` as the first parameter will start searching right after the previously returned token. You can change the set of delimiters (*wcs2*) from call to call.

Returns

wcstok function returns a pointer to the first wide character of the token, or a null pointer if there is no token. In later calls with the same token string, wcstok returns a pointer to the next token in the string. When there are no more tokens, wcstok returns `NULL`.

Example Code

This example uses wcstok to locate the tokens in the wide-character string `str1`.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   static wchar_t str1[] = L"?a??b,,,#c";
   static wchar_t str2[] = L"\t \t";
   wchar_t *t *ptr1, *ptr2;

   t = wcstok(str1, L"?", &ptr1);             /* t points to the token  L"a"   */
    printf("t = '%ls'\n", t);
   t = wcstok(NULL, L",", &ptr1);             /* t points to the token L"?b"   */
    printf("t = '%ls'\n", t);
   t = wcstok(str2, L" \t,", &ptr2);          /* t is a null pointer
    printf("t = '%ls'\n", t);
   t = wcstok(NULL, L"#,", &ptr1);           /* t points to the token L"c"    */
   printf("t = '%ls'\n", t);
   t = wcstok(NULL, L"?", &ptr1);            /* t is a null pointer           */
   printf("t = '%ls'\n", t);
   return 0;

   /*****************************************************************************
      The output should be similar to :

          t = 'a'
          t = '?b'
          t = '(null)'
          t = 'c'
          t = '(null)'
   ****************************************************************************/
}
```

Related Information

- strtok

----

wcstol - Convert Wide-Character to Long Integer

wcstol - Convert Wide-Character to Long Integer

Syntax

```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

Description

wcstol converts the wide-character string pointed to by *nptr* to a long integer value. *nptr* points to a sequence of wide characters that can be interpreted as a numerical value of type `long int`. wcstol stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the *base* .

When you use wcstol, *nptr* should point to a string with the following form:

```
>>                                                          ><
       white-space     +     0       digits
                             0x
                               0X
```

If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is `0`, the prefix determines the base (8, 16, or 10): the prefix `0` means base 8 (octal); the prefix `0x` or `0X` means base 16 (hexadecimal); using any other digit without a prefix means decimal.

The behavior of wcstol is affected by the LC_CTYPE category of the current locale.

Returns

wcstol returns the converted long integer value. If no conversion could be performed, wcstol returns `0`. If the correct value is outside the range of representable values, wcstol returns LONG_MAX or LONG_MIN returned (according to the sign of the value), and sets errno to ERANGE.

If the string *nptr* points to is empty or does not have the expected form, no conversion is performed, and the value of *nptr* is stored in the object pointed to by *endptr* , provided that *endptr* is not a null pointer.

Example Code

This example uses wcstol to convert the wide-character string `wcs` to a long integer value.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"10110134932";
   wchar_t *stopwcs;
   long     l;
   int      base;

   printf("wcs = \"%ls\"\n", wcs);
   for (base=2; base<=8; base*=2) {
      l = wcstol(wcs, &stopwcs, base);
      printf("   wcstol = %ld\n"
             "   Stopped scan at \"%ls\"\n\n", l, stopwcs);
   }
   return 0;

   /****************************************************************************
      The output should be similar to :

      wcs = "10110134932"
         wcstol = 45
         Stopped scan at "34932"

         wcstol = 4423
         Stopped scan at "4932"

         wcstol = 2134108
         Stopped scan at "932"
   ****************************************************************************/
}
```

Related Information

- atol
- scanf
- strtol
- strtold
- strtoul
- wcstod
- wcstoul

----

wcstombs - Convert Wide-Character String to Multibyte String

wcstombs - Convert Wide-Character String to Multibyte String

Syntax

```
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *string, size_t n);
```

Description

wcstombs converts the wide-character string pointed to by *string* into the multibyte array pointed to by *dest* . The conversion stops after *n* bytes in *dest* are filled or after a wide null character is encountered.

Only complete multibyte characters are stored in *dest* . If the lack of space in *dest* would cause a partial multibyte character to be stored, wcstombs stores fewer than *n* bytes and discards the incomplete character.

The behavior of wcstombs is affected by the LC_CTYPE category of the current locale.

Returns

If successful, wcstombs returns the number of bytes converted and stored in *dest* , not counting the terminating null character. The string pointed to by *dest* ends with a null character unless wcstombs returns the value *n* .

If it encounters an invalid wide character, wcstombs returns `(size_t)-1`.

If the area pointed to by *dest* is too small to contain all the wide characters represented as multibyte characters, wcstombs returns the number of bytes containing complete multibyte characters.

If *dest* is a null pointer, the value of *len* is ignored and wcstombs returns the number of elements required for the converted wide characters.

Example Code

In this example, wcstombs converts a wide-character string to a multibyte character string twice. The first call converts the entire string, while the second call only converts three characters. The results are printed each time.

```
#include <stdio.h>
#include <stdlib.h>

#define  SIZE          20

int main(void)
{
   char dest[SIZE];
   wchar_t *dptr = L"string";
   size_t count = SIZE;
   size_t length;

   length = wcstombs(dest, dptr, count);
   printf("%d characters were converted.\n", length);
   printf("The converted string is \"%s\"\n\n", dest);

   /* Reset the destination buffer                                            */

   memset(dest, '\0', sizeof(dest));

   /* Now convert only 3 characters                                           */

   length = wcstombs(dest, dptr, 3);
   printf("%d characters were converted.\n", length);
   printf("The converted string is \"%s\"\n", dest);
   return 0;

   /*****************************************************************************
      The output should be:

      6 characters were converted.
      The converted string is "string"

      3 characters were converted.
      The converted string is "str"
   *****************************************************************************/
}
```

Related Information

- mbstowcs
- wcslen
- wctomb

----

wcstoul - Convert Wide-Character String to Unsigned Long

wcstoul - Convert Wide-Character String to Unsigned Long

Syntax

```
#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);
```

Description

wcstoul converts the wide-character string pointed to by *nptr* to an unsigned long integer value. *nptr* points to a sequence of wide characters that can be interpreted as a numerical value of type `unsigned long int`. wcstoul stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the *base* .

When you use wcstoul, *nptr* should point to a string with the following form:

```
>>                                                  ><
       white-space     0      digits
                       0x
                       0X
```

If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is `0`, the prefix determines the base (8, 16, or 10): the prefix `0` means base 8 (octal); the prefix `0x` or `0X` means base 16 (hexadecimal); using any other digit without a prefix means decimal.

Additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed and wcstoul stores the value of *nptr* in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The behavior of wcstoul is affected by the LC_CTYPE category of the current locale.

Returns

wcstoul returns the converted unsigned long integer value. If no conversion could be performed, wcstoul returns `0`. If the correct value is outside the range of representable values, wcstoul returns ULONG_MAX and sets errno to ERANGE.

If the string *nptr* points to is empty or does not have the expected form, no conversion is performed, and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Example Code

This example uses wcstoul to convert the string `wcs` to an unsigned long integer value.

```
#include <stdio.h>
#include <wchar.h>

#define BASE 2

int main(void)
{
   wchar_t *wcs = L"1000e13 camels";
   wchar_t *endptr;
   unsigned long int answer;

   answer = wcstoul(wcs, &endptr, BASE);
   printf("The input wide string used: '%ls'\n"
          "The unsigned long int produced: %lu\n"
          "The substring of the input wide string that was not"
          " converted to unsigned long: '%ls'\n", wcs, answer, endptr);
   return 0;

   /****************************************************************************
      The output should be similar to :

      The input wide string used: '1000e13 camels'
      The unsigned long int produced: 8
      The substring of the input wide string that was not converted to
      unsigned long: 'e13 camels'
   ****************************************************************************/
}
```

Related Information

- strtod
- strtol
- strtold
- wcstod
- wcstol

----

wcswcs - Locate Wide-Character Substring

wcswcs - Locate Wide-Character Substring

Syntax

```
#include <wchar.h>
```

```
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

Description

`wcswcs` locates the first occurrence of *string2* in the wide-character string pointed to by *string1* . In the matching process, `wcswcs` ignores the `wchar_t` null character that ends *string2* .

Returns

`wcswcs` returns a pointer to the located string or `NULL` if the string is not found. If *string2* points to a string with zero length, `wcswcs` returns *string1* .

Example Code

This example finds the first occurrence of the wide character string `pr` in `buffer1`.

```
#include <stdio.h>
#include <wchar.h>

#define  SIZE          40

int main(void)
{
   wchar_t buffer1[SIZE] = L"computer program";
   wchar_t *ptr;
   wchar_t *wch = L"pr";

   ptr = wcswcs(buffer1, wch);
   printf("The first occurrence of %ls in '%ls' is '%ls'\n", wch, buffer1, ptr);
   return 0;

   /****************************************************************************
      The output should be:

      The first occurrence of pr in 'computer program' is 'program'
   ****************************************************************************/
}
```

Related Information

- strchr
- strcspn
- strpbrk
- strrchr
- strspn
- wcschr
- wcscspn
- wcspbrk
- wcsrchr
- wcsspn

----

wcswidth - Determine Display Width of Wide-Character String

wcswidth - Determine Display Width of Wide-Character String

Syntax

```
#include <wchar.h>
int wcswidth (const wchar_t *wcs, size_t n);
```

Description

wcswidth determines the number of printing positions occupied on a display device by a graphic representation of *n* wide characters in the string pointed to by *wcs* (or fewer than *n* wide characters, if a null wide character is encountered before *n* characters have been exhausted). The number is independent of its location on the device.

The behavior of wcswidth is affected by the LC_CTYPE category.

Returns

wcswidth returns the number of printing positions occupied by the wide-character string. If *wcs* points to a null wide character, wcswidth returns 0. If any wide character in *wcs* is not a printing character, wcswidth returns -1.

Example Code

This example uses wcswidth to calculate the display width of ABC.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"ABC";

   printf("wcs has a width of: %d\n", wcswidth(wcs,3));
   return 0;

   /*****************************************************************************
      The output should be similar to :

      wcs has a width of: 3
   *****************************************************************************/
}
```

Related Information

- wcwidth

----

wcsxfrm - Transform Wide-Character String

wcsxfrm - Transform Wide-Character String

Syntax

```
#include <wchar.h>
size_t wcsxfrm(wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

Description

wcsxfrm transforms the wide-character string pointed to by *wcs2* to values that represent character collating weights and places the resulting wide-character string into the array pointed to by *wcs1* . The transformation is determined by the program's locale. The transformed string is not necessarily readable, but can be used with the wcscmp function.

The transformation is such that if wcscmp were applied to two transformed wide-character strings, the results would be the same as applying the wcscoll function to the two corresponding untransformed strings.

No more than *n* elements are placed into the resulting array pointed to by *wcs1* , including the terminating null wide character. If *n* is 0, *wcs1* can be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The behavior of wcsxfrm is controlled by the LC_COLLATE category of the current locale.

Returns

wcsxfrm returns the length of the transformed wide-character string (excluding the terminating null wide character). If the value returned is *n* or more, the contents of the array pointed to by *wcs1* are indeterminate.

If *wcs1* is a null pointer, wcsxfrm returns the number of elements required to contain the transformed wide string.

If *wcs2* contains invalid wide characters, wcsxfrm returns (`size_t`)-1. The transformed values of the invalid characters are either less than or greater than the transformed values of valid wide characters, depending on the option chosen for the particular locale definition.

If *wcs2* contains wide characters outside the domain of the collating sequence. wcsxfrm sets errno to EILSEQ.

Example Code

This example uses wcsxfrm to transform two different strings with the same collating weight. It then uses wcscmp to compare the new strings.

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
#include <wchar.h>

int main(void)
{
   wchar_t *string1 = L"stride ng1";
   wchar_t *string2 = L"stri*ng1";
   wchar_t *newstring1, *newstring2;
   int  length1, length2;

   if (NULL == setlocale(LC_ALL, "Fr_FR")) {
      printf("setlocale failed.\n");
      exit(EXIT_FAILURE);
   }
   length1 = wcsxfrm(NULL, string1, 0);
   length2 = wcsxfrm(NULL, string2, 0);
   if (NULL == (newstring1 = calloc(length1 + 1, sizeof(wchar_t))) ||
       NULL == (newstring2 = calloc(length2 + 1, sizeof(wchar_t)))) {
      printf("insufficient memory\n");
      exit(EXIT_FAILURE);
   }
   if ((wcsxfrm(newstring1, string1, length1 + 1) != length1) ||
       (wcsxfrm(newstring2, string2, length2 + 1) != length2)) {
      printf("error in string processing\n");
      exit(EXIT_FAILURE);
   }
   if (0 != wcscmp(newstring1, newstring2))
      printf("wrong results\n");
   else
      printf("correct results\n");
   return 0;

   /****************************************************************************
      The output should be similar to :

      correct results
   ****************************************************************************/
}
```

Related Information

- strxfrm
- wcscmp
- wcscoll

----

wctob - Convert Wide Character to Byte

wctob - Convert Wide Character to Byte

Syntax

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t wc);
```

Description

wctob determines whether *wc* corresponds to a member of the extended character set, whose multibyte character has a length of 1 byte

The behavior of wctob is affected by the LC_CTYPE category of the current locale.

Returns

If *c* corresponds to a multibyte character with a length of 1 byte, wctob returns the single-byte representation. Otherwise, wctob returns EOF.

Example Code

This example uses wctob to test if the wide character A is a valid single-byte character.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wint_t wc = L'A';

   if (wctob(wc) == wc)
      printf("%lc is a valid single byte character\n", wc);
   else
      printf("%lc is not a valid single byte character\n", wc);
   return 0;

   /****************************************************************************
      The output should be similar to :

      A is a valid single byte character
   ****************************************************************************/
}
```

Related Information

- mbtowc
- wctomb
- wcstombs

----

wctomb - Convert Wide Character to Multibyte Character

wctomb - Convert Wide Character to Multibyte Character

Syntax

```
#include <stdlib.h>
int wctomb(char *string, wchar_t wc);
```

Description

wctomb converts the wide character *wc* into a multibyte character and stores it in the location pointed to by *string* . wctomb stores a maximum of `MB_CUR_MAX` characters in *string* .

The behavior of wctomb is affected by the LC_CTYPE category of the current locale.

Returns

wctomb returns the length in bytes of the multibyte character. If *wc* is not a valid multibyte character, wctomb returns -1.

If *string* is a `NULL` pointer, wctomb returns `0`.

Note: On platforms that support shift states, wctomb can also return a nonzero value to indicate that the multibyte encoding is state dependent.

Example Code

This example calls wctomb to convert the wide character `c` to a multibyte character.

```
#include <stdio.h>
#include <stdlib.h>

#define  SIZE          40

int main(void)
{
   static char buffer[SIZE];
   wchar_t wch = L'c';
   int length;

   length = wctomb(buffer, wch);
   printf("The number of bytes that comprise the multibyte ""character is %i\n",
      length);
   printf("And the converted string is \"%s\"\n", buffer);
   return 0;

   /****************************************************************************
      The output should be:

      The number of bytes that comprise the multibyte character is 1
      And the converted string is "c"
   ****************************************************************************/
}
```

Related Information

- mbtowc
- wcslen
- wcstombs

----

wctype - Get Handle for Character Property Classification

wctype - Get Handle for Character Property Classification

Syntax

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

wctype returns a handle for the specified character *class* from the LC_CTYPE category. You can then use this handle with the iswctype function to determine if a given wide character belongs to that *class* .

The following strings correspond to the standard (basic) character classes or properties:

```
cols='15 15 15 15'.

  "
  "alnum"
  "alpha"
  "blank" "

  "
  "cntrl"
  "digit"
  "graph" "

  "
  "lower"
  "print"
  "punct" "

  "
  "space"
  "upper"
  "xdigit" "
```

These classes are described in isalnum to isxdigit and iswalnum to iswxdigit.

You can also give the name of a user-defined class, as specified in a locale definition file, as the *property* argument.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale.

Returns

wctype returns a value of type `wctype_t` that represents the property and can be used in calls to iswctype. If the given *property* is not valid for the current locale (LC_CTYPE category), wctype returns `0`.

Values returned by wctype are valid until a call to setlocale that modifies the LC_CTYPE category.

Example Code

This example uses wctype to return each standard property, and calls iswctype to test a wide character against each property.

```
#include <wctype.h>
#include <stdio.h>

#define UPPER_LIMIT 0xFF

int main(void)
{
   int wc;

   for (wc = 0; wc <= UPPER_LIMIT; wc++) {
      printf("%#4x ", wc);
      printf("%lc", iswctype(wc, wctype("print")) ? (wchar_t)wc : ' ');
      printf("%s", iswctype(wc, wctype("alnum"))  ? " AN" : "   ");
      printf("%s", iswctype(wc, wctype("alpha"))  ? " A " : "   ");
      printf("%s", iswctype(wc, wctype("blank"))  ? " B " : "   ");
      printf("%s", iswctype(wc, wctype("cntrl"))  ? " C " : "   ");
      printf("%s", iswctype(wc, wctype("digit"))  ? " D " : "   ");
      printf("%s", iswctype(wc, wctype("graph"))  ? " G " : "   ");
      printf("%s", iswctype(wc, wctype("lower"))  ? " L " : "   ");
      printf("%s", iswctype(wc, wctype("punct"))  ? " PU" : "   ");
      printf("%s", iswctype(wc, wctype("space"))  ? " S " : "   ");
      printf("%s", iswctype(wc, wctype("print"))  ? " PR" : "   ");
      printf("%s", iswctype(wc, wctype("upper"))  ? " U " : "   ");
      printf("%s", iswctype(wc, wctype("xdigit")) ? " X " : "   ");
      putchar('\n');
   }
   return 0;

   /****************************************************************************
      The output should be similar to :
       :
      0x1f            C
      0x20         B                 S  PR
      0x21 !                G     PU    PR
      0x22 "                G     PU    PR
      0x23 #                G     PU    PR
      0x24 $                G     PU    PR
      0x25 %                G     PU    PR
      0x26 &                G     PU    PR
      0x27 '                G     PU    PR
      0x28 (                G     PU    PR
      0x29 )                G     PU    PR
      0x2a *                G     PU    PR
      0x2b +                G     PU    PR
      0x2c ,                G     PU    PR
      0x2d -                G     PU    PR
      0x2e .                G     PU    PR
      0x2f /                G     PU    PR
      0x30 0 AN          D  G           PR    X
      0x31 1 AN          D  G           PR    X
      0x32 2 AN          D  G           PR    X
      0x33 3 AN          D  G           PR    X
      0x34 4 AN          D  G           PR    X
      0x35 5 AN          D  G           PR    X
       :
   ****************************************************************************/
}
```

Related Information

- iswalnum to iswxdigit
- iswctype

----

wcwidth - Determine Display Width of Wide Character

wcwidth - Determine Display Width of Wide Character

Syntax

```
#include <wchar.h>
int wcwidth (const wint_t wc);
```

Description

wcwidth determines the number of printing positions that a graphic representation of *wc* occupies on a display device. Each of the printing wide characters occupies its own number of printing positions on a display device. The number is independent of its location on the device.

The behavior of wcwidth is affected by the LC_CTYPE category.

Returns

wcwidth returns the number of printing positions occupied by *wc*. If *wc* is a null wide character, wcwidth returns 0. If *wc* is not a printing wide character, *wc* returns -1.

Example Code

This example determines the printing width for the wide character A.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wint_t wc = L'A';

   printf("%lc has a width of %d\n", wc, wcwidth(wc));
   return 0;

   /****************************************************************************
      The output should be similar to :

      A has a width of 1
   ****************************************************************************/
}
```

Related Information

- wcswidth

----

write - Writes from Buffer to File

write - Writes from Buffer to File

Syntax

```
#include <io.h>
int write(int handle, const void *buffer, unsigned int count);
```

Description

write writes *count* bytes from *buffer* into the file associated with *handle* . The write operation begins at the current position of the file pointer associated with the given file. If the file is open for appending, the operation begins at the end of the file. After the write operation, the file pointer is increased by the number of bytes actually written.

If the given file was opened in text mode, each line feed character is replaced with a carriage return/line feed pair in the output. The replacement does not affect the return value.

Returns

write returns the number of bytes moved from the buffer to the file. The return value may be positive but less than *count* . A return value of `-1` indicates an error, and `errno` is set to one of the following values: compact break=fit.

Value	**Meaning**
EBADF	The file handle is not valid, or the file is not open for writing.
ENOSPC	No space is left on the device.
EOS2ERR	The call to the operating system was not successful.

Example Code

This example writes the contents of the character array `buffer` to the output file whose handle is `fh`.

```
#include <io.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

#define FILENAME  "write.dat"

int main(void)
{
   int fh;
   char buffer[20];

   memset(buffer, 'a', 20);                 /* initialize storage             */
   printf("\nCreating " FILENAME ".\n");
   system("echo Sample Program > " FILENAME);
   if (-1 == (fh = open(FILENAME, O_RDWR|O_APPEND))) {
      perror("Unable to open " FILENAME);
      return EXIT_FAILURE;
   }
   if (5 != write(fh, buffer, 5)) {
      perror("Unable to write to " FILENAME);
      close(fh);
      return EXIT_FAILURE;
   }
   printf("Successfully appended 5 characters.\n");
   close(fh);
   return 0;

   /****************************************************************************
      The program should create a file "write.dat" containing:

      Sample Program
      aaaaa

      The output should be:

      Creating write.dat.
      Successfully appended 5 characters.
   ****************************************************************************/
}
```

Related Information

- creat
- fread
- fwrite
- lseek
- open
- read
- _sopen
- _tell

Notices

September 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "(C) (your company name) (year). All rights reserved."

(C)Copyright International Business Machines Corporation 1996. All rights reserved.
Note to U.S. Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

> IBM Director of Licensing
> IBM Corporation
> 500 Columbus Avenue
> Thornwood, NY 10594
> U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:
BookManager
C/2
IBM
Operating System/2
OS/2
PM
Presentation Manager
SAA
Systems Application Architecture
VisualAge
Workplace Shell

The following terms are trademarks of other companies:

C++	AT&T, Inc.
Intel	Intel Corporation
Open Software Foundation	Open Software Foundation, Inc.
OSF	Open Software Foundation, Inc.
PCMCIA	Personal Computer Memory Card International Association
Unicode	Unicode Inc.

Windows is a trademark of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary defines terms and abbreviations from:

- The *IBM Dictionary of Computing* , New York: McGraw-Hill, copyright 1994 by International Business Machines Corporation. Copies may be purchased from McGraw-Hill or in bookstores.
- *American National Dictionary for Information Systems* , ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- The ANSI/EIA Standard-440-A: *Fiber Optic Terminology* . Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington DC 20006. Definitions are identified by the symbol (E) after the definition.
- *The Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- *X/Open CAE Specification. Commands and Utilities, Issue 4. July, 1992* . These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990* . These definitions are indicated by the symbol *ISO.1* after the definition.

A

----

abstraction (data)

abstraction (data)

A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

----

access

access

An attribute that determines whether or not a class member is accessible in an expression or declaration.

----

access mode

access mode

1. A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. (A).
2. The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM* .
3. A particular form of access permitted to a file. *X/Open* .

----

alignment

alignment

The storing of data in relation to certain machine-dependent boundaries. *IBM* .

American National Standards Institute

American National Standards Institute

See ANSI.

ANSI (American National Standards Institute)

ANSI (American National Standards Institute)

An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. (A).

API (application program interface)

API (application program interface)

A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM* .

----

application

application

1. The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM* .

2. A collection of software components used to perform specific types of user-oriented work on a computer. *IBM* .

----

application program

application program

A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM* .

----

argument

argument

1. A parameter passed between a calling program and a called program. *IBM* .
2. In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter* .
3. In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open* .

----

array

array

In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. *IBM* .

----

array element

array element

A data item in an array. *IBM* .

ASCII (American National Standard Code for Information Interchang

ASCII (American National Standard Code for Information Interchange)

The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM* .

Note: IBM has defined an extension to ASCII code (characters 128-255).

B

----

backslash

backslash

The character \. This character is named <backslash> in the portable character set.

----

binary stream

binary stream

1. An ordered sequence of untranslated characters.

2. A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM* .

----

blank character

blank character

1. A graphic representation of the space character. (A).

2. A character that represents an empty position in a graphic character string. (T).

3. One of the characters that belongs to the *blank* character class as defined via the LC_CTYPE category in the current locale. *X/Open* .

----

block

block

1. In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. (I).

2. A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. (T).

3. The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

----

boundary alignment

boundary alignment

The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM* .

----

brackets

brackets

The characters [(left bracket) and] (right bracket), also known as *square brackets* . When used in the phrase "enclosed in (square) brackets" the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open* .

----

built-in

built-in

1. A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline.

2. In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. (I). Synonymous with predefined. *IBM* .

C

C++ class library

C++ class library

See *class library* .

C++ library

C++ library

A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

----

call

call

To transfer control to a procedure, program, routine, or subroutine. *IBM* .

----

cast

cast

In the C and C++ languages, an expression that converts the type of the operand to a specified data type (the operator). *IBM* .

----

character

character

1. A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. (A).

2. A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of the multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open* . *ISO.1* .

----

character array

character array

An array of type char. *X/Open* .

----

character class

character class

A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open* .

----

character constant

character constant

1. A constant with a character value. *IBM* .

2. A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM* .

3. In some languages, a character enclosed in apostrophes. *IBM* .

----

character set

character set

1. A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. (T).
2. All the valid characters for a programming language or for a computer system. *IBM* .
3. A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM* .
4. See also *portable character set* .

----

character string

character string

A contiguous sequence of characters terminated by the first null byte and including the first null byte. *X/Open* .

----

child

child

A node that is subordinate to another node in a tree structure. Only the root node is not a child.

----

class

class

1. A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members.
2. A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

----

class library

class library

A collection of C++ classes.

C library

C library

A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM* .

----

code page

code page

1. An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code.
2. A particular assignment of hexadecimal identifiers to graphic characters.

----

codeset

codeset

Synonym for code element set. *IBM* .

----

collating element

collating element

The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open* .

----

collating sequence

collating sequence

1. A specified arrangement used in sequencing. (I). (A).
2. An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. (A).
3. The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

----

collection

collection

1. An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection.
2. In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library

Collection Class Library

A set of classes that provide basic functions for collections, and can be used as base classes.

----

command

command

A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

----

condition

condition

A relational expression that can be evaluated to a value of either true or false. *IBM* .

----

const

const

1. An attribute of a data object that declares the object cannot be changed.
2. A keyword that allows you to define a variable whose value does not change.

----

constant

constant

1. In programming languages, a language object that takes only one specific value. (I).
2. A data item with a value that does not change. *IBM* .

----

control character

control character

1. A character whose occurrence in a particular context specifies a control function. (T).
2. Synonymous with nonprinting character. *IBM* .
3. A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open* .

----

conversion

conversion

1. In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. (I).
2. The process of changing from one method of data processing to another or from one data processing system to another. *IBM* .
3. The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM* .
4. A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

----

conversion descriptor

conversion descriptor

A per-process unique value used to identify an open codeset conversion. *X/Open* .

----

coordinated universal time (UTC)

coordinated universal time (UTC)

Equivalent to Greenwich Mean Time (GMT)

----

current working directory

current working directory

1. A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open* . *ISO.1* .
2. The directory that is searched when a file name is entered with no indication of the directory that lists the file name. The operating system assumes that the current directory is the root directory unless a path to another directory is specified. *IBM* .
3. In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM* .

D

----

data object

data object

1. A storage area used to hold a value.
2. Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays.
3. In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM* .

----

data stream

data stream

A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM* .

----

data type

data type

The properties and internal representation that characterize data.

DBCS (double-byte character set)

DBCS (double-byte character set)

A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM* .

----

declaration

declaration

1. In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM* .
2. Establishes the names and characteristics of data objects and functions used in a program.

----

default locale

default locale

1. The C locale, which is always used when no selection of locale is performed.
2. A system default locale, named by locale-related environmental variables.

----

define directive

define directive

A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

----

definition

definition

1. A data description that reserves storage and may provide an initial value.
2. A declaration that allocates storage, and may initialize a data object or specify the body of a function.

----

delete

delete

1. A C++ keyword that identifies a free storage deallocation operator.
2. A C++ operator used to destroy objects created by `new`.

----

device

device

A computer peripheral or an object that appears to the application as such. *X/Open* . *ISO.1* .

----

directory

directory

A type of file containing the names and controlling information for other files or other directories. *IBM* .

----

display

display

To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open* .

----

dot

dot

The file name consisting of a single dot character (.). *X/Open* . *ISO.1* .

----

double-byte character set

double-byte character set

See *DBCS* .

----

double-precision

double-precision

Pertaining to the use of two computer words to represent a number in accordance with the required precision. (I). (A).

----

dump

dump

To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM* .

----

dynamic

dynamic

Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM* .

E

EBCDIC (extended binary-coded decimal interchange code)

EBCDIC (extended binary-coded decimal interchange code)

A coded character set of 256 8-bit characters. *IBM* .

E-format

E-format

Floating-point format, consisting of a number in scientific notation. *IBM* .

----

element

element

The component of an array, subrange, enumeration, or set.

----

empty string

empty string

1. A string whose first byte is a null byte. Synonymous with null string. *X/Open* .
2. A character array whose first element is a null character. *ISO.1* .

----

epoch

epoch

The time zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time. *X/Open* . *ISO.1* .

----

exception

exception

1. Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception).
2. In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. (I).

----

exception handler

exception handler

1. Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications.
2. A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM* .

----

executable file

executable file

A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open* .

----

extension

extension

1. An element or function not included in the standard language.
2. File name extension.

F

----

file mode

file mode

An object containing the *file mode bits* and file type of a file, as described in <sys/stat.h>. *X/Open* .

----

file mode bits

file mode bits

A file's file permission bits, set-user-ID-on-execution bit (S_ISUID) and set-group-ID-on-execution bit (S_ISGID). *X/Open* .

----

file scope

file scope

A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

----

for statement

for statement

A looping statement that contains the word *for* followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

----

function

function

A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM* .

----

function call

function call

An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM* .

G

----

global

global

Pertaining to information available to more than one program or subroutine. *IBM* .

----

global variable

global variable

A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time)

GMT (Greenwich Mean Time)

The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinate universal time (UTC).

Greenwich Mean Time

Greenwich Mean Time

See GMT.

H

----

header file

header file

A text file that contains declarations used by a group of functions, programs, or users.

----

heap

heap

An unordered flat collection that allows duplicate elements.

I

I18N

I18N

Abbreviation for *internationalization* . The abbreviation uses the first character of the word internationalization (i), the last character (n), and omitting the middle eighteen characters.

----

identifier

identifier

1. One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. (A).
2. In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function.

(A).

3. A sequence of letters, digits, and underscores used to identify a data object or function. *IBM* .

----

if statement

if statement

A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM* .

----

include directive

include directive

A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

----

include file

include file

See *header file* .

----

input stream

input stream

A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM* .

----

instance

instance

An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration:

```
box box1, box2;
```

----

instruction

instruction

A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

----

internationalization

internationalization

The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open* .

Synonymous with *I18N* .

----

iteration

iteration

The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

K

----

key access

key access

A property that allows elements to be accessed by matching keys.

----

keyword

keyword

1. A predefined word reserved for the C and C++ languages, that may not be used as an identifier.
2. A symbol that identifies a parameter in JCL.

L

----

label

label

An identifier within or attached to a set of data elements. (T).

----

library

library

1. A collection of functions, calls, subroutines, or other data. *IBM* .

2. A set of object modules that can be specified in a link command.

----

link

link

To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

----

linker

linker

A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM* .

----

literal

literal

1. In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, "APRIL" represents the string of characters APRIL, 3.0005E2 represents the number 300.05. (I).

2. A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM* .

3. A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM* .

----

local

local

1. In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. (I).

2. Pertaining to that which is defined and used only in one subdivision of a computer program. (A).

----

locale

locale

The definition of the subset of a user's environment that depends on language, territory (country), and code set information. *X/Open* .

----

lvalue

lvalue

An expression that represents a data object that can be both examined and altered.

M

----

macro

macro

An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

----

mask

mask

A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. (I). (A).

----

member

member

A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

----

method

method

In the C++ language, a synonym for member function.

----

migrate

migrate

To move to a changed operating environment, usually to a new release or version of a system. *IBM* .

----

mode

mode

A collection of attributes that specifies a file's type and its access permissions. *X/Open* . *ISO.1* .

----

module

module

A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

----

multibyte code set (MBCS)

multibyte code set (MBCS)

A code set that needs one or more bytes to represent a single character.

----

multitasking

multitasking

A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. (I). (A).

N

----

name

name

In the C++ language, a name is commonly referred to as an identifier. However, syntactically, a name can be an identifier, operator function name, conversion function name, destructor name or qualified name.

NULL

NULL

In the C and C++ languages, a pointer that does not point to a data object. *IBM* .

----

null character (NUL)

null character (NUL)

The ASCII or EBCDIC character '\0' with the hex value `00`, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named<NUL> in the portable character set.

----

null pointer

null pointer

The value that is obtained by converting the number 0 into a pointer; for example, `(void *) 0`. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open* .

----

null string

null string

1. A string whose first byte is a null byte. Synonymous with *empty string* . *X/Open* .
2. A character array whose first element is a null character. *ISO.1* .

----

null value

null value

A parameter position for which no value is specified. *IBM* .

O

----

object

object

1. A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance* .)
2. In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class* . *IBM* .
3. An instance of a class.

----

open file

open file

A file that is currently associated with a file descriptor. *X/Open* . *ISO.1* .

----

operating system (OS)

operating system (OS)

Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

----

operator precedence

operator precedence

In programming languages, an order relation defining the sequence of the application of operators within an expression. (I).

----

overflow

overflow

1. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.
2. That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM* .

P

----

parameter

parameter

1. In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open* .

2. Data passed between programs or procedures. *IBM* .

----

parent process

parent process

1. The program that originates the creation of other processes by means of `spawn` or `exec` function calls. See also *child process* .

2. A process that creates other processes.

----

path name

path name

1. A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes will be treated as a single slash. The interpretation of the path name is described in *pathname resolution* . *ISO.1* .

2. A file name specifying all directories leading to the file.

----

period

period

The character (**.**). The term *period* is contrasted against *dot* , which is used to describe a specific directory entry. This character is named <period> in the portable character set.

----

pointer

pointer

In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM* .

----

portable character set

portable character set

The set of characters specified in POSIX 1003.2, section 2.4 and in ISO 646 IRV. It includes uppercase A-Z, lowercase a-z, 0-9, and basic punctuation characters.

----

portability

portability

The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

----

positional parameter

positional parameter

A parameter that must appear in a specified location relative to other positional parameters. *IBM* .

----

precedence

precedence

The priority system for grouping different types of operators with their operands.

----

predefined macros

predefined macros

Frequently used routines provided by an application or language for the programmer.

----

preprocessor

preprocessor

A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

----

printable character

printable character

One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open* .

----

process

process

1. An instance of an executing application and the resources it uses.

2. An address space and single thread of control that executes within that address space, and its required system resources. *X/Open* . *ISO.1* .

----

process ID

process ID

The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t* .) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open* . *ISO.1* .

----

public

public

Pertaining to a class member that is accessible to all functions.

R

----

redirection

redirection

In the shell, a method of associating files with the input or output of commands. *X/Open* .

----

reentrant

reentrant

The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

----

regular expression

regular expression

1. A mechanism to select specific strings from a set of character strings.
2. A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern.
3. A string containing wildcard characters and operations that define a set of one or more possible strings.

----

root

root

In the OS/2 operating system, the base directory.

----

run-time library

run-time library

A compiled collection of functions whose members can be referred to by an application program during run time execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The run-time library itself is not statically bound into the application modules.

S

----

scope

scope

1. That part of a source program in which a variable is visible.
2. That part of a source program in which an object is defined and recognized.

----

semaphore

semaphore

An object used by multithread applications for signalling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

----

sequence

sequence

A sequentially ordered flat collection.

----

session

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see &setsid.. There can be multiple process groups in the same session. *X/Open* . *ISO.1* .

----

signal

signal

1. A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero.

2. A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open* . *ISO.1* .

----

signal handler

signal handler

A function to be called when the signal is reported.

----

space character

space character

The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open* .

----

specifiers

specifiers

Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

----

standard error (stderr)

standard error (stderr)

An output stream usually intended to be used for diagnostic messages. *X/Open* .

----

standard input (stdin)

standard input (stdin)

1. An input stream usually intended to be used for primary data input. *X/Open* .
2. The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM* .

----

standard output (stdout)

standard output (stdout)

1. An output stream usually intended to be used for primary data output. *X/Open* .
2. In the AIX operating system, the primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM* .

----

statement

statement

An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

----

static

static

A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

----

stream

stream

1. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.
2. A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the fdopen or fopen functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open* .

----

stream buffer

stream buffer

A stream buffer is a buffer between the ultimate consumer, ultimate producer, and the I/O Stream Library functions that format data. It is implemented in the I/O Stream Library by the `streambuf` class and the classes derived from `streambuf`.

----

string

string

A contiguous sequence of bytes including and terminated by the first null byte. *X/Open* .

----

string constant

string constant

Zero or more characters enclosed in double quotation marks.

----

struct

struct

An aggregate of elements, having arbitrary types.

----

structure

structure

A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

----

subsystem

subsystem

A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. (T).

----

support

support

In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM* .

T

----

text file

text file

A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}-which is defined in limits.h-bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). *X/Open* .

----

this

this

A C++ keyword that identifies a special type of pointer in a member function, that references the class object with which the member function was invoked.

----

thread

thread

The smallest unit of operation to be performed within a process. *IBM* .

----

tilde

tilde

The character ~. This character is named <tilde> in the portable character set.

----

token

token

The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM* .

----

trap

trap

An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred. (I).

----

type

type

The description of the data and the operations that can be performed on or by the data. See also *data type* .

----

type conversion

type conversion

Synonym for *boundary alignment* .

----

type definition

type definition

A definition of a name for a data type. *IBM* .

----

type specifier

type specifier

Used to indicate the data type of an object or function being declared.

U

----

underflow

underflow

1. A condition that occurs when the result of an operation is less than the smallest possible nonzero number.
2. Synonym for arithmetic underflow, monadic operation. *IBM* .

----

union

union

1. In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM* .

2. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

V

----

variable

variable

In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. (I).

W

----

while statement

while statement

A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM* .

----

white space

white space

1. Space characters, tab characters, form-feed characters, and new-line characters.

2. A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open* .

----

wide character

wide character

A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

----

wide-character string

wide-character string

A contiguous sequence of wide-character codes including and terminated by the first null wide-character code. *X/Open* .

----

working directory

working directory

Synonym for *current working directory* .
